R. Grum / R. Hund

DAS GROSSE BASIC-LEXIKON zum



Eine Edition aus dem Heim-Verlag

R. Grum / R. Hund

DAS GROSSE BASIC-LEXIKON zum



Eine Edition aus dem Heim-Verlag

Das große Basic-Lexikon zum Schneider CPC 464
R. Grum / R. Hund
– 1. Auflage – Darmstadt: Heim, 1984
ISBN 3-923250-15-0

beim Heim-Verlag · Organisation + Datentechnik
Heidelberger Landstr. 194 · 6100 Darmstadt
Telefon 0 61 51 - 5 53 75

Druck: Druckerei der Heim OHG, 6100 Darmstadt.

Vorwort

Mit dem vorliegenden Lexikon zum Schneider-Mikrocomputer ⟩⟩CPC 464⟨⟨ erfüllt der Heim-Verlag den Wunsch vieler Leser des STANDARD-BASIC-Buches nach einem Nachschlagewerk, aus dem eingehende Informationen über die Anwendung der einzelnen BASIC-Kommandos *mit dazugehörigen Beispielen* entnommen werden können. Neben der Beschreibung der Kommandos liegt hier der Akzent auf den Beispielen; durch Beispiele können oftmals schwer verständliche Sachverhalte klarer und eindeutiger als durch viele Worte beschrieben werden. Anfänger und Fortgeschrittene haben somit die Möglichkeit, gezielt nachzuschlagen und sich umfassend zu informieren.

Herr Raimund Grum, einer der Autoren des BASIC-STANDARD-Buchs, hat mit viel Eifer und Sorgfalt das vorliegende Lexikon geschaffen und gleichzeitig die darin enthaltenen Programme auf einer Programmcassette zusammengestellt, die es dem Programmierer ermöglicht, Beispielprogramme in den Schneider-Computer zu laden und gezielt zu einzelnen BASIC-Kommandos entsprechende Übungen durchzuführen. Sie können diese Programmübungscassette mit der Nr. C-205 entweder bei Ihrem Fachhändler oder unmittelbar beim Heim-Verlag, Heidelberger Landstraße 194, 6100 Darmstadt bestellen.

Im übrigen sei hier noch auf ein weiteres Buch aus dem Heim-Verlag verwiesen, das in Kürze unter dem Titel "BASIC-Programmierhilfe zum Schneider-Computer⟩⟩CPC464⟨⟨"erscheint. Das Buch ist vor allem für den Anfänger eine wertvolle Einstiegshilfe, kann aber auch dem Fortgeschrittenen als Nachschlagewerk von Nutzen sein.

Sollten Sie irgendwelche fachlichen Fragen im Zusammenhang mit dem Lexikon oder der Programmübungscassette haben, so wenden Sie sich bitte schriftlich an die Redaktion des Heim-Verlags, Binger Str. 10, 6100 Darmstadt.

Mit freundlichen Grüßen
Ihre
Redaktion des Heim-Verlags

Darmstadt, im November 1984

Inhaltsverzeichnis

Seite

Einführung 1

Programmieren mit BASIC 1
Der BASIC-Interpreter 2
Der BASIC-Compiler 3
Interpreter und Compiler im Wechsel 4
Programmstellung 5
BASIC-Sprachelemente 7
Eingabe- und Verarbeitungsarten 8
Eingabeformat einer Programmzeile 9
Format der Anweisungen 9

Lexikon der BASIC-Kommandos von A bis Z 11–270

Fehlermeldungen 271–294

Einführung

Programmieren in BASIC

BASIC ist ein Kunstwort, das aus den Anfangsbuchstaben der nachstehenden englischen Worte zusammengesetzt ist:

Beginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

zu deutsch "symbolischer Allzweckbefehlscode für Anfänger".

BASIC gehört zu dem Kreis der problemorientierten Programmiersprachen wie etwa COBOL, FORTRAN und ALGOL, wobei BASIC eine gewisse Ähnlichkeit mit FORTRAN besitzt.

BASIC wurde Anfang der sechziger Jahre im Dartmouth College in den USA von den Professoren John. G. Kemeny und Thomas E. Kurtz geschaffen. Das angestrebte Ziel der beiden Sprachschöpfer war es, eine

* leicht zu erlernende
* interaktive und
* leicht anzuwendende

Programmiersprache zu gestalten, die der allgemeinen Programmierung dienen sollte.

BASIC läßt sich für mathematisch-technische und kaufmännische Programme, für einfache und komplizierte Unterhaltungsspiele, für Textbe- und -verarbeitung, prinzipiell für jede nur denkbare Art von Programmen verwenden.

BASIC erlaubt einen Dialogbetrieb mit dem Computer und ist hierbei so flexibel, daß vorhandene Programme ohne großen zusätzlichen Programmieraufwand geändert und somit auf den spezifischen Anwendungsfall angepaßt werden können.

BASIC ist eine problemorientierte, aber eine einfach zu erlernende Programmiersprache. Kenntnisse von irgendwelchen Maschinensprachen sind nicht erforderlich. Kommandos bzw. Anweisungen an den Computer werden in knappen, englischen Worten eingegeben. Die Programme sind dabei "transportierbar", d.h., sie sind auf den unterschiedlichsten Computermodellen und -fabrikaten lauffähig. Es kommt also nicht darauf an, welchen Computer man einsetzt. Allerdings gilt diese Aussage mit gewissen Einschränkungen. BASIC hat im Verlaufe von 20 Jahren über 200 Dialekte entwickelt, so daß man folgendes feststellen kann:

In der einen BASIC-Version gibt es Kommandos, die in der anderen Version fehlen und umgekehrt.

Oftmals werden unterschiedliche Worte für ein- und dieselbe Funktion benutzt.

Die Umwandlung von einem BASIC-Dialekt in einen anderen ist oftmals nicht besonders schwierig, jedoch recht lästig.

Ihr ⟨⟨ CPC 464 ⟩⟩ besitzt aber einen sehr umfangreichen BASIC- Wortschatz, der aus einem weitverbreiteten BASIC-Dialekt stammt und zusätzlich über einige sehr nützliche und interessante Graphik- und Sound-Anweisungen verfügt.

BASIC-Interpreter

Die meisten BASIC-Sprachprozessoren sind Interpreter. Unter einem Interpreter versteht man ein Programm, das BASIC-Kommandos in die Maschinensprache des Mikrocomputers übersetzt. Jeder Mikrocomputertyp hat seinen eigenen BASIC-Interpreter; der Benutzer merkt davon jedoch nichts, er sieht nur ein- und dieselbe BASIC- Sprache!

Die Arbeitsweise des BASIC-Interpreters:

Jedesmal, wenn in einem Programm eine BASIC-Anweisung durchlaufen wird, muß der Interpreter sie in die Maschinensprache übersetzen. Wird eine Anweisung 1000 mal durchlaufen, wird sie auch 1000 mal übersetzt, d.h. interpretiert.

BASIC-Compiler

Anders geht ein BASIC-Compiler vor:

Ein Compiler übersetzt das gesamte Programm in die Maschinensprache, bevor es ausgeführt wird. Auf diese Weise wird jede BASIC-Anweisung nur einmal übersetzt, und zwar ohne Rücksicht darauf, wie oft die Anweisung im Programm vorkommt. Es ist deshalb leicht einzusehen, daß ein compiliertes Programm schneller läuft als ein interpretiertes.

Ein weiterer Vorteil des Compilers ist es, daß er beim Umwandeln das gesamte Programm auf die Einhaltung der sprachlichen Regeln überprüft und Syntax-Fehler, d.h., fehlende Kommandos, fehlerhaft geschriebene Befehlswörter meldet. Die erkannten Fehler werden ausgegeben, so daß der Programmierer sie in Ruhe beseitigen kann.

Der Interpreter dagegen meldet nur dann einen Fehler, wenn er bei der Programmausführung einen Fehler in einer Programmzeile findet. Ein scheinbar fehlerfreies Programm kann einige Male ohne Fehlermeldung gelaufen sein, bis es dann plötzlich zu einer Fehlermeldung kommt, die von einer vorher nicht benutzten Programmzeile ausgeht. Wird ein Fehler angetroffen, so wird das Programm vom Interpreter abgebrochen, d.h., es ist dann "abgestürzt".

Allerdings hat das Arbeiten mit dem Compiler auch seine Nachteile:

Zum einen benötigt der Compiler mehr Platz im Arbeitsspeicher des Computers, da im Speicher sowohl Platz für den Quellcode, das BASIC-Programm, als auch für den Objekt-Code, das Maschinenprogramm, vorhanden sein muß. Zum anderen macht jede kleine und kleinste Programmänderung in einem compilierten Programm eine erneute und zeitaufwendige Compilierung erforderlich. Viel leichter und schneller lassen sich Änderungen durchführen, wenn man mit dem BASIC-Interpreter arbeitet. Hier schreibt man eine fehlerhafte Anweisung neu und startet danach das Programm erneut. Bequemer geht es nicht!

Interpreter und Compiler im Wechsel

Es gibt aber auch einen Weg, die Vorteile von Interpreter und Compiler gemeinsam zu nutzen:

* Zuerst wird das Programm mit dem Interpreter erstellt und ausgetestet.

* Anschließend wird es mit dem Compiler bearbeitet, d.h. es wird compiliert.

* Programmänderungen werden mit dem Interpreter ausgeführt und wiederum ausgetestet.

* Danach wird das Programm wieder compiliert.

Auf diese Weise kann man in der ersten Phase des Programmierens das Programm sehr schnell ändern, sowie unter dem Interpreter testen und besitzt nach der Compilierung ein schneller laufendes Programm als bei einer Interpreterversion.

In welcher Sprache wird programmiert?

Um Mißverständnisse zu vermeiden:

Programmiert wird immer in BASIC.

Lediglich die Ausführung des Programms im Computer unterscheidet sich in der Ausführungsart als

* Interpreterfassung oder

* compilierte Fassung.

Mit dem Compilieren von BASIC-Programmen sollten Sie sich erst beschäftigen, wenn Ihnen BASIC geläufig ist.

Programmerstellung

Die Programmerstellung gliedert sich grundsätzlich immer in die Phasen:

1. Problemanalyse

   1.1 Betrachtung der Fakten und Teilprobleme, die den gesamten Umfang des zu lösenden Problems exakt bestimmen sollen.

   1.2 Bestimmung der Datenströme, die zu vorgegebenen Zeitpunkten koordiniert ablaufen bzw. vorliegen müssen, damit ein sinnvoller Ablauf gewährleistet werden kann.

2. Systemanalyse

   2.1 Die Ausarbeitung der dem Problem adäquaten Programmlogik, die die grundlegenden Strukturelemente (wie z.B. Wiederholungen, Folge, Auswahl) ausweist, d.h., die Zergliederung in Teilaufgaben unter Berücksichtigung des notwendigen Detailierungsgrades.

   2.2 Dokumentation der deduzierten Programmstruktur z.B. durch Struktogramme oder Programmablaufplänen.

3. Programmierung

   3.1 Deduzieren der Programmiersprache, z.B. BASIC, FORTRAN oder PASCAL. Die genannten Programmiersprachen besitzen für bestimmte Problemfälle sowohl Vor- als auch Nachteile bei anderen Anwendungen.

   3.2 Generieren des Programmtextes entsprechend der vorgegebenen Syntax der Programmiersprache.

   3.3 Einbeziehung von Modulen aus der Programmbibliothek.

4. Testen des Programms

   Verifikation bzw. Falsifikation der über das Programm angestrebten Problemlösung.

5. Dokumentieren des Programms

   Hier ist die umfassende Enddokumentation des Programms angesprochen.

6. Programmanwendung

Die LOGIK des PROGRAMMIERENS und das DOKUMENTIEREN von PROGRAMMEN werden in den Erkäuterungen zu den einzelnen Anweisungen zwar deutlich, sind aber nicht explizit ausgeführt. Diese Themen finden Sie aufbereitet in einem weiteren Band aus dem HEIM-Verlag.

Das Lexikon als Nachschlagewerk soll Ihnen, auch über Ihre ersten eigenen BASIC-Programme hinaus, eine andauernde Hilfe bei der Programmerstellung sein.

Der Schwerpunkt des Ihnen vorliegenden Bandes wurde daher auf die Vermittlung aller BASIC-Anweisungen des ⟩⟩ CPC464 ⟨⟨ Interpreters in ihrer korrekten Syntax und die Darstellung der Anwendung gelegt, so daß eine Übertragung auf einen vorgegebenen Anwendungsfall für Sie unproblematisch wird.

Der Umgang mit einer neuen Sprache bedarf einiger Grundkenntnisse über ihren Aufbau, um danach effektiv einen Dialog mit dem ⟩⟩ CPC 464 ⟨⟨ zu eröffnen. Die nachstehenden Kurzerläuterungen zu der Sprache werden in den weiteren Kapiteln als bekannt vorausgesetzt und dienen Ihnen als Einführung in die Sprache bzw. zur Wiederholung von Gewußtem.

BASIC-Sprachelemente

Die künstlich geschaffene Computersprache BASIC besitzt wie jede natürliche Sprache eine Grammatik und einen ihr eigenen charakteristischen Zeichensatz.

Der Zeichensatz umfaßt alphabetische Zeichen, numerische Zeichen und Sonderzeichen:

*Buchstaben: A, B, C, ..., Z, a, b, c, ..., z

*Zahlen ......: 0, 1, 2, ..., 9

*Sonderzeichen: # + * = etc.

Die addierende Kombination dieser Zeichen ergibt die Vielfalt der unter BASIC verfügbaren Sprachelemente:

*BASIC-Schlüsselwörter für:

- Arithmetische Ausdrücke
- Standardfunktionen

- Operatoren
- Deklarationsanweisungen
- Eingabeanweisungen
- Ausgabeanweisungen
- Zuweisungen
- Organisatorische Anweisungen
- Datensicherungs- und Speicherungsanweisungen
- Sonderfunktionen

*Numerische Konstanten
*Stringkonstanten
*Numerische Variablen
*Stringvariablen
*Feldvariablen

Die aufgeführten Elemente bilden, zusammengefügt zu Anweisungen und Anweisungsketten, die Basis eines BASIC-Programms, die von dem eingesetzten Rechner in der vorgegebenen Reihen- und Strukturfolge interpretiert und abgearbeitet werden.

Eingabe- und Verarbeitungsmodi

Prinzipiell werden bei dem BASIC-Interpreter zwei Arten der Eingabe unterschieden:

1. Direkt-Modus

   Hierbei werden BASIC-Anweisungen ohne Zeilennummer eingegeben und sofort nach Drücken der ENTER-Taste ausgeführt. Die Ergebnisse von arithmetischen oder logischen Operatoren werden unmittelbar nach der Eingabe und Drücken der ENTER-Taste ausgegeben.

2. Indirekt-Modus:

   Es werden Programmzeilen mit Zeilennummern eingegeben, die dann im Arbeitsspeicher verfügbar bleiben. Die so präsenten Anweisungsfolgen können mit der RUN-Anweisung zur Ausführung gebracht werden.

Eingabe-Format einer Programmzeile

Jede Programmzeile eines BASIC-Programms besitzt ein gleichbleibendes Format:

nnnn BASIC-Anweisung [:BASIC-Anweisung ...]⟨ENTER⟩

Die Programmzeile beginnt immer mit der Zeilen- bzw. Anweisungsnummer und kann eine Länge bis zu 255 Zeichen aufweisen. Wurden mehrere Anweisungen zu einer Anweisungskette in einer Zeile zusammengefaßt, so sind die einzelnen Anweisungen durch einen Doppelpunkt zu trennen. Das Ende einer Programmzeile ist durch Drücken der ENTER-Taste zu kennzeichnen.

Format der Anweisungen

Die BASIC-Sprachelemente (Statements) besitzen eine eigene Syntax, die über die Formatangabe der einzelnen Anweisungen erkennbar ist. Die Formatangabe von BASIC-Anweisungen ist hierbei so zu interpretieren:

Anweisungen, Befehle bzw. Kommandos sind in Großbuchstaben angegeben. Der Interpreter des ⟩⟩ CPC 464 ⟨⟨ fordert die Eingabe in Großbuchstaben nicht. Es ist aber zweckmäßiger sie so einzugeben, da die Programmübersicht hier durch wesentlich verbessert wird.

Ergänzungen zu den Anweisungen, Befehlen bzw. Kommandos sind mit den Zeichen ⟨ ⟩ versehen und die geforderten Angaben sind in Kleinbuchstaben aufgeführt.

Angaben in eckigen Klammern [ ] stellen Optionen dar. Sie können bei Bedarf wahlweise zur Basis-Anweisung an- bzw. zugefügt werden.
Andere Sonderzeichen, die nicht den Zeichen ⟨ ⟩ und [ ] entsprechen, sind entsprechend der Formatangabe auch so anzugeben. Kommatas, Semikola, runde Klammern etc. können zum notwendigen Bestandteil einer BASIC-Anweisung ge hören.

Kann ein unmittelbar vorausgehender Teil einer BASIC- Anweisung definiert oft wiederholt werden, so wird dieses durch Fortsetzungspunkte ... angegeben.

Nach einer BASIC-Anweisung kann ein blank (Leerzeichen) notwendigerweise folgen, ehe die nachfolgenden Parameter nachgestellt werden dürfen. Im Einzelfall ist dieses anhand der Formatangabe zu prüfen.

Format: ABS(⟨numerischer Ausdruck⟩)

Zweck:

Ermitteln des Absolutbetrages eines numerischen Ausdrucks.

Anwendung:

ABS(X)

Hierbei steht die Variable X für einen numerischen Ausdruck. Die Anweisung ergibt den absoluten Wert oder "Betrag" einer Variablen oder eines Ausdrucks, d.h. positive Ausdrücke bleiben unverändert, negative ergeben positive.

Beispiel:

```
10 MODE 1
20 A=-4
30 LOCATE 1,5
40 PRINT "Numerischer Ausdruck      Absoluter Wert"
50 PRINT STRING$(39," ")
60 PRINT "      4                    ";ABS(4)
70 PRINT "     19                    ";ABS(19)
80 PRINT "     A=-4                  ";ABS(A)
90 PRINT "     -3*4                  ";ABS(-3*4)
```

Ergebnis:

Es werden die absoluten Werte in einer Tabelle ausgegeben.

Vergleichen Sie hierzu auch: SGN

AFTER

Format: AFTER ⟨Zeitangabe⟩[,⟨Timer⟩] GOSUB ⟨Zeilennummer⟩

Zweck:

Aufrufen von Unterprogrammen in bestimmten Zeitabständen.

Anwendung:

Mit AFTER kann nach einer vorprogrammierten ⟨Zeitangabe⟩ ein Unterprogramm automatisch angesprungen werden.

⟨Zeitangabe⟩=1 entspricht 0,02 Sekunden. Nach Ablauf der ⟨Zeitangabe⟩ wird das Unterprogramm aufgerufen.

⟨Timer⟩ ist einer der 4 Zeitgeber, anwählbar mit den Werten 0 bis 3. Wird die Angabe ⟨Timer⟩ nicht genutzt, dann ist ⟨Timer⟩ auf 0 gesetzt. Die Zeitgeber besitzen unterschiedliche Prioritäten. ⟨Timer⟩ gleich 3 besitzt die höchste, ⟨Timer⟩ gleich 0 die niedrigste Priorität.

Die ⟨Zeilennummer⟩ gibt die Anfangszeile eines Unterprogramms an, das mit RETURN beendet werden muß.

Die bei der AFTER-Anweisung benutzten ⟨Timer⟩ sind identisch mit denen der EVERY-Anweisung. Dieses ist zu beachten, da durch eine später folgende EVERY-Anweisung der angegebene ⟨Timer⟩ überschrieben wird. Dieses gilt auch für die umgekehrte Anweisungsfolge.

Beispiel:

```
5 MODE 2
10 AFTER 10,1 GOSUB 100
20 FOR I=0 TO 50:PRINT "*";:NEXT I
30 END
100 PRINT "Zeit abgelaufen und zum Unterprogramm
gesprungen"
110 RETURN
```

Ergebnis:

```
******************************************
Zeit abgelaufen und zum Unterprogramm gesprungen
*********
```

Vergleichen Sie hierzu auch: EVERY, REMAIN

AND

Logische Verknüpfung: AND

Zweck:

Ableiten von Entscheidungen zur Programmausführung.

Anwendung:

Die logische Verknüpfung a AND b ist genau dann wahr, wenn die Ausdrücke "a" und "b" wahr sind.

Hinweis: "AND" ist keine Addition von zwei Variablen !

Wahrheitstafel: WAHR = 1; FALSCH = 0

a	b	a AND b
0	0	0
1	0	0
0	1	0
1	1	1

Beispiel:

```
10 MODE 1
20 A=1
30 B=2
40 C$="TEST"
50 IF A=1 AND C$="TEST" THEN 60 ELSE 80
60 PRINT "In Variable A ist die Zahl 1"
70 PRINT " und in C$ ist das Wort 'TEST'"
80 IF A=5 AND B=10 THEN END
90 A=5
100 B=10
110 GOTO 40
```

Ergebnis:

```
In Variable A ist die Zahl 1
und in C$ ist das Wort 'TEST'
```

Vergleichen Sie hierzu auch: OR, NOT

ASC

Format: ASC(⟨String⟩)

Zweck:

Ermitteln des ASCII-Wertes vom ersten Zeichen einer Zeichenkette.

Anwendung:

ASC(A$) liefert den Zahlenwert (ASCII-WERT) des Zeichens **A$** oder des ersten Zeichens der Zeichenkette **A$**.

Beispiel:

```
10 MODE 2
20 A$="Zeichenkette"
30 B$="Z"
40 PRINT "ASCII-Werte"
50 PRINT STRING$(60,"")
60 PRINT
70 PRINT ASC("k"),ASC(A$),ASC(B$),ASC("Z"),ASC("z")
80 LOCATE 5,8
90 PRINT "Programmabbruch mit ESC-Taste!"
100 GOTO 100
```

Ergebnis:

```
ASCII-Werte
____________________________________________________________
107          90           90           90           122

    Programmabbruch mit ESC-Taste
```

Vergleichen Sie hierzu auch: CHR$

ATN

Format: ATN(⟨numerischer Ausdruck⟩)

Zweck:

Berechnen des Arcustangens eines numerischen Ausdrucks.

Anwendung:

ATN berechnet den Arcustangens von dem vorgegebenen ⟨numerischen Aus druck⟩. Die Variable ⟨numerischer Ausdruck⟩ ist dann ein Winkel in Grad, wenn zuvor mit der Anweisung DEG auf Grad umgestellt wurde. Der ⟨numerischer Ausdruck⟩ ist ein Winkel im Bogenmaß (Radiant), wenn zuvor mit der Anweisung RAD auf Bogenmaß eingestellt wurde. Wurde keine der beiden Anweisungen eingegeben, so wird die Berechnung im Bogenmaß vorgenommen.

Beispiel:

```
10 MODE 2
20 X=1
30 PRINT ATN(X),
40 DEG
50 PRINT ATN(X),
60 RAD
70 PRINT ATN(X)
80 END
```

Ergebnis:

```
0.785398163              45              0.785398163
```

Vergleichen Sie hierzu auch: DEG, RAD, SIN, COS, TAN

AUTO

Format: `AUTO [⟨Zeilennummer⟩][,⟨Schrittweite⟩]`

Zweck:

Automatisches Erzeugen von Zeilennummern.

Anwendung:

Die `AUTO`-Anweisung ohne Angabe der ⟨Zeilennummer⟩ und der ⟨Schrittweite⟩ bewirkt eine automatische Zeilennumerierung mit der Anfangszeilennummer 10 und der Schrittweite 10.

Die Angabe der ⟨Zeilennummer⟩ bestimmt die Anfangszeilennummer, die ⟨Schrittweite⟩ die Schrittweite der automatischen Zeilennumerierung. Wird ⟨Schrittweite⟩ nicht vorgegeben, so wird die Schrittweite 10 angenommen. Entfällt die Angabe für ⟨Zeilennummer⟩, so lautet die Anfangszeilennummer 10.

Beispiele:

`AUTO`	erzeugt die Zeilennummern 10,20,30,...
`AUTO 230`	erzeugt die Zeilennummern 230,240,250,...
`AUTO ,5`	erzeugt die Zeilennummern 10,15,20,25,...
`AUTO 183,4`	erzeugt die Zeilennummern 183,187,191,...

Vergleichen Sie hierzu auch: `RENUM`

Format: `BIN$(Integerausdruck[,Integerausdruck])`

Zweck:

Der Befehl `BIN$` wandelt einen Integerausdruck in eine Binärzahl um.

Anwendung:

```
BIN(X,Y)
```

Hierbei ist X eine Zahl, die kleiner als 65536 oder &FFFF sein muß. Y ist das Format, in das die Binärzahl gesetzt wird. Zahlen mit kleinerer Stellenzahl als das vorgegebene Format werden von Nullen angeführt. Ist das Format kleiner als die Binärzahl, so bestimmt die Länge der Binärzahl das Format.

Beispiel:

```
10 MODE 2
20 PRINT BIN$(68),BIN$(68,8),BIN$(68,10),BIN$(68,2)
30 A=715
40 PRINT BIN$(A)
50 END
```

Ergebnis:

```
1000100   1000101   0001000100   1000100   1011001011
```

Vergleichen Sie hierzu auch: `HEX$`, `DEC$`, `STR$`

BORDER

Format: BORDER ⟨Farbcode⟩[,⟨Farbcode⟩]

Zweck:

Setzen der Farbe des Bildschirmrandes.

Anwendung:

Die BORDER-Anweisung setzt die Farbe des Bildschirmrandes auf die Farbe mit dem entsprechenden ⟨Farbcode⟩ und erlaubt den Wechsel zwischen den zwei Farben, die mit dem ersten und zweiten ⟨Farbcode⟩ angegeben wurden.

Die Zahl des ⟨Farbcode⟩s muß zwischen 0 und 26 liegen und definiert die Farbe nach der Tabelle.

Farbcode-Tabelle

⟨Farbcode⟩	Farbe	⟨Farbcode⟩	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Beispiel:

`BORDER 9` setzt den Bildschirmrand auf die Farbe grün

Wird zusätzlich die Zahl für den zweiten ⟨Farbcode⟩ angegeben, so wechseln die Farben des Bildschirmrandes zwischen diesen beiden Farben in wählbaren Zeitintervallen.

Beispiel:

`BORDER 11,24` setzt den Bildschirmrand abwechselnd himmelblau und hellgelb.

Vergleichen Sie hierzu auch: `SPEED INK, PAPER, PEN, INK, WINDOW`

CALL

Format: CALL ⟨Startadresse⟩[,⟨Variablenliste⟩]

Zweck:

Aufrufen eines Maschinensprache-Unterprogramms.

Anwendung:

Die CALL-Anweisung verzweigt in eine Assembler-Routine, d.h., ein in Maschinensprache geschriebenes Unterprogramm wird von BASIC aufgerufen. Die Variablenliste muß nicht angegeben sein.

Vergleichen Sie hierzu auch: PEEK und POKE

Format: `CAT`

Zweck:

Erstellen einer Namensliste aller auf Kassette gefundenen Dateien.

Anwendung:

Die `CAT`-Anweisung liest die Kassette und erstellt ein Inhaltsverzeichnis, d.h., es gibt die Namen der abgespeicherten Programme, die Programmblöcke und das Aufzeichnungsformat der Programme und Dateien auf dem Monitor aus. Mit diesem Befehl werden keine Programme eingelesen. Lesen Sie möglichst jedes abgespeicherte Programm vor einer `NEW`-Anweisung oder Abschalten des ⟩⟩ CPC 464 ⟨⟨. Ist Ihr Programm in einer lesbaren Qualität abgespeichert worden, so erhalten Sie durch die `CAT`-Anweisung ein OK.

Das im Arbeitsspeicher vorhandene Programm wird hierbei nicht beeinflußt, so daß Sie, falls kein OK erscheint, dieses Programm noch einmal abspeichern können.

Auf dem Monitor erscheint:

```
Dateiname/Programmname   Blocknummer   Kennzeichen
```

Das Kennzeichen beschreibt das Aufzeichnungsformat nach folgendem Schlüssel:

$ = BASIC-Programme
% = geschütztes BASIC-Programm
*= ASCII-Datei
+ = binärer Datenblock

Wurde mit SAVE "" ein Programm abgespeichert, so erscheint der Hinweis:

    "Unnamed file block ⟩X⟩⟨"

auf dem Monitor.

Vergleichen Sie hierzu auch:

RUN "⟨Dateiname⟩", LOAD, SAVE, MERGE, CHAIN MERGE, CHAIN

CHAIN

Format:

CHAIN "Programmname"[,⟨Zeilennummer⟩]
bzw.

CHAIN MERGE "Programmname"[,Zeilennummer⟩]

[,DELETE ⟨Zeilennummer-Zeilennummer⟩]

Zweck:

Aufrufen bzw. Einmischen eines Programms von Kassette mit Übergabe der Variablen des aufrufenden Programms.

Anwendung:

Die CHAIN-Anweisung liest ein aus einem anderen laufenden Programm aufgerufenes Programm vom Datacorder in den Arbeitsspeicher ein. Bei dieser Anweisung werden alle Variablen an das aufgerufene Programm übergeben und das alte Programm überschrieben.

Die Angabe ⟨Zeilennummer⟩ bewirkt, daß das nachgeladene Programm ab der ⟨Zeilennummer⟩ startet, d.h., die Zeile mit der ⟨Zeilennummer⟩ enthält die erste Anweisung für die Programmausführung.

Die CHAIN MERGE-Anweisung liest ein aus einem laufenden Programm aufgerufenes Programm vom Datacorder ein, ohne hierbei das im Arbeitsspeicher vorhandene Programm zu löschen. Die Angabe ⟨Zeilennummer⟩ ist auch hierbei wieder die Startzeile des zusammengemischten Programms. Wird diese nicht angegeben, so wird ab der ersten Zeile begonnen. Sollen Programm-Module nach dem Zusammenmischen gelöscht werden, um z.B. den vorhandenen Arbeitsspeicherplatz zu vergrößern, so ist mit DELETE ⟨Zeilennummer⟩-⟨Zeilennummer⟩ ein bestimmter Zeilenbereich löschbar.

Die Variablen werden entsprechend der CHAIN-Anweisung übergeben. Verzichten Sie auf die Angabe des Programmnamens, so lädt BASIC das erste gefundene Programm, was häufig zu interessanten, aber leider nicht lauffähigen Programmen führt.

Wird CHAIN mit der MERGE-Anweisung genutzt, so werden alle benutzereigenen Definitionen für Variablen und Funktionen unwirksam. So die vorgenommenen Definitionen DEFINT, DEFREAL und DEFSTR, die dann unwirksam sind, d.h., sie müssen im "neuen" Programm neu angelegt werden.

Beachten Sie bei den CHAIN- und CHAIN-MERGE-Anweisungen, daß geschützte Programme nicht eingemischt werden können.

Vergleichen Sie hierzu auch: RUN, GOTO, MERGE, LOAD, SAVE

Format: `PRINT CHR$(⟨ASCII-Wert⟩)`

Zweck:

Ausgeben von Steuerzeichen, Zeichen und Graphikzeichen.

Anwendung:

`PRINT CHR$(⟨ASCII-Wert⟩)` dient der Ausgabe von ASCII-Zeichen mit dem ⟨ASCII-Wert⟩ und einer Zeichenkettenlänge von 1. Der ⟨ASCII-Wert⟩ besitzt bei darstellbaren Zeichen Werte von 32 bis 255. Dieser Befehl dient z.B. auch der Druckeransteuerung, wobei ⟨ASCII-Wert⟩ hier auch Werte kleiner 32 annehmen kann.

Beispiel:

```
10 FOR N=65 TO 90
20 PRINT CHR$(N);
30 NEXT N
40 END
```

Ergebnis:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Für einige Anwendungsfälle ist es wünschenswert, einen kurzen Signalton, ähnlich dem einer Schreibmaschine, auszugeben. Dieses kann mit der Anweisung:

```
PRINT CHR$(7)
```

in einfacher Form realisiert werden.

Andere ASCII-Werte bewirken mit der PRINT-Anweisung besondere Ausgabe formen und Ausgabeformate.

Hierzu ein Anwendungsbeispiel:

```
10 BORDER 14
20 MODE 1
30 INK 2,19
40 LOCATE 8,2
50 PRINT ">> CPC 464 <<"
60 LOCATE 8,4
70 PRINT "Zeichnet ohne zu loeschen!"
80 LOCATE 5,12
90 PRINT "TRANSPARENT"
100 PRINT CHR$(22)+CHR$(1)
110 LOCATE 5,15
120 ORIGIN 10,165
130 DRAW 600,0,2
140 FOR I=1 TO 16
150 READ A$
160 PRINT A$+" ";:FOR K=1 TO 500: NEXT K
170 NEXT I
180 DATA T,R,A,N,S,P,A,R,E,N,T,M,O,D,U,S
190 PRINT CHR$(22)+CHR$(0)
200 LOCATE 5,18
210 ORIGIN 10,120
220 DRAW 600,0,3
230 FOR I=1 TO 1000:NEXT I
240 FOR K=1 TO 11
250 READ B
260 PRINT CHR$(B)+CHR$(32);:FOR J=1 TO 500:NEXT J
270 NEXT K
280 DATA 78,79,82,77,65,76,77,79,68,85,83
290 GOTO 290
```

In Zeile 100 wird der "Transparent-Modus" mit

```
PRINT CHR$(22)+CHR$(1)
```

eingeschaltet und in Zeile 190 mit

```
PRINT CHR$(22)+CHR$(0)
```

ausgeschaltet. Den Effekt können Sie durch den Programmablauf bestens selbst erkennen.

Mit der `READ`-Anweisung in Zeile 250 werden die ASCII-Werte eingelesen, die in der `DATA`-Zeile 280 zur Verfügung gestellt werden. Diese werden dann in Zeile 260 mit `PRINT CHR$(B)` ausgegeben. `CHR$(32)` entspricht einem Leerzeichen, so daß zwischen den einzelnen Buchstaben ein Blank ausgegeben wird.

Weitere ASCII-Werte und ihre Bedeutungen finden Sie in der ASCII- Tabelle.

Vergleichen Sie hierzu auch:

`ASC, LEFT$, RIGHT$, MID$, STR$, TAB, SPACE$, SPC`

CINT

Format: CINT(<numerischer Ausdruck>)

Zweck:

Umwandeln einer Zahl in einen gerundeten Integer-Wert.

Anwendung:

CINT(<numerischer Ausdruck>) rundet den <numerischen Ausdruck> auf einen ganzzahligen Wert. Die Anweisung CINT(<numerischer Ausdruck>) entspricht in ihrer Wirkung dem Befehl ROUND(<numerischer Ausdruck>,0).

Beispiel:

```
? cint(-2.58)
```

Ergebnis:

```
-3
```

Vergleichen Sie hierzu auch: INT, FIX, CREAL, ROUND

Format: `CLEAR`

Zweck:

Löschen aller Variablen.

Anwendung:

`CLEAR` setzt alle numerischen Variablen auf Null und löscht alle Stringvariablen. Strings erhalten damit die Länge Null.

Vergleichen Sie hierzu auch: `NEW, CLS, CLG`

CLG

Format: CLG [⟨Code⟩]

Zweck:

Löschen des Grapik-Bildschirmfensters und der Graphik-Cursorposition.

Anwendung:

Mit der CLG-Anweisung wird das Graphik-Bildschirmfenster gelöscht. Wenn die Angabe von ⟨Code⟩ erfolgt, wird das Bildschirmfenster vollständig in der dem ⟨Code⟩ entsprechenden Farbe eingefärbt. Die Farbe ist abhängig von der Formatanweisung MODE. Wurde keine Angabe für ⟨Code⟩ vorgegeben, so wird der Graphik-Bildschirm mit dem letzten durch CLG verwendeten ⟨Code⟩ gelöscht.

Wurde vorher kein Graphik-Bildschirmfenster mit der ORIGIN-Anweisung definiert, so löscht die CLG-Anweisung das aktuelle Textbildschirm fenster.

Beispiel:

```
10 MODE 1
20 REM ** DAS TEXTBILDSCHIRMFENSTER WIRD FARBIG
UMRANDET **
30 BORDER 14
40REM**EINGRAPHIKBILDSCHIRMFENSTERWIRDAUFGEBAUT**
50 ORIGIN 0,0,50,550,300,100
60 REM ** DAS GRAPHIKBILDSCHIRMFENSTER WIRD
EINGEFAERBT **
70 CLG 10
80 END
```

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟩⟨24	20	1
15*	= Farbcode	16⟩⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch: `ORIGIN`, `CLS`

Format: `CLOSEIN`

Zweck:

Schließen eines Eingabefiles.

Anwendung:

Die `CLOSEIN`-Anweisung schließt einen Eingabefile vom Datacorder, der vorab mit `OPENIN` eröffnet wurde. Ohne vorherige `CLOSEIN`-Anweisung kann kein neuer Eingabefile mit OPENIN eröffnet werden. Ein Anwendungsbeispiel finden Sie unter `OPENIN` und `OPENOUT`.

Vergleichen Sie hierzu auch: `OPENIN, CLOSEOUT`

CLOSEOUT

Format: CLOSEOUT

Zweck:

Schließen der eröffneten Ausgabedatei.

Anwendung:

Die CLOSEOUT-Anweisung schließt einen Ausgabefile vom Datacorder, der vorab mit der OPENOUT-Anweisung eröffnet wurde. Ein neues Ausgabefile kann erst nach vorheriger CLOSEOUT-Anweisung eröffnet werden. Die Daten werden vor der Ausgabe auf dem Datacorder in einen Puffer von 2k Bytes Länge eingelesen. Ist dieser Puffer gefüllt oder wird die CLOSEOUT-Anweisung im Programm gegeben, so wird der Pufferinhalt blockweise abgespeichert. Beachten Sie hierbei, daß erst durch die CLOSEOUT-Anweisung die "Restdaten" aus dem Puffer abgespeichert werden.

Vergleichen Sie hierzu auch: OPENOUT, CLOSEIN

Format: CLS [# ⟨Datenstrom⟩]

Zweck:

Löschen eines Bildschirmfensters.

Anwendung:

CLS steht für "CLEAR SCREEN" und löscht alle Zeichen auf dem Monitor. Wird die Angabe #⟨Datenstrom⟩ vorgegeben, so färbt CLS das entsprechende Bildschirmfenster ein.

Die Angabe # ⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Höhere Werte für die Angabe # Datenstrom führen zu der Fehlermeldung "Improper argument".

Vergleichen Sie hierzu auch: CLG, CLEAR, NEW

CONT

Format: CONT

Zweck:

Fortsetzen eines abgebrochenen Programmablaufs.

Anwendung:

Mit der CONT-Anweisung wird eine Programmunterbrechnung nach den Anweisungen BREAK, STOP oder END aufgehoben und der Programmablauf fortgesetzt. Die Fortsetzung des Programms erfolgt an der gleichen Stelle, an der es abgebrochen wurde.

Die CONT-Anweisung wirkt nicht, wenn das Programm während der Unterbrechung verändert wurde, oder aber bei einer INPUT$-Anweisung abgebrochen wurde. Hier wird dann die Fehlermeldung "Cannot CONTinue" ausgegeben, d.h., das Programm kann nicht fortgesetzt werden.

Die CONT-Anweisung wird immer nur im Kommando-Modus benutzt und hat ihre Berechtigung bei dem Austesten von BASIC-Programmen.

Vergleichen Sie hierzu auch: STOP, END

COS

Format: COS(⟨numerischer Ausdruck⟩)

Zweck:

Berechnet den Cosinus.

Anwendung:

COS(⟨numerischer Ausdruck⟩) berechnet den Cosinus vom vorgegebenen Ausdruck im Bogenmaß. Der ⟨numerische Ausdruck⟩ist dann ein Winkel in Grad, wenn zuvor mit dem Befehl DEG auf Grad umgestellt wurde. Mit RAD kann wieder auf Bogenmaß zurückgestellt werden.

Beispiel:

```
10 MODE 2
20 X=60
30 PRINT COS(X),
40 DEG
50 PRINT COS(X),
60 RAD
70 PRINT COS(X)
80 END
```

Ergebnis:

```
-0.952412983          0.5          -0.952412983
```

Vergleichen Sie hierzu auch: DEG, RAD, SIN, TAN, ATN

CREAL

Format: CREAL(⟨numerischer Ausdruck⟩)

Zweck:

Umwandeln von Integerzahlen in Gleitkommazahlen.

Anwendung:

Die CREAL-Anweisung wandelt den ⟨numerischen Ausdruck⟩in eine Gleitkommazahl um. Falls ein Compiler eingesetzt werden soll, müssen die Variablentypen übereinstimmen. Dieses ist bei alleinigem Einsatz des BASIC-Interpreters nicht erforderlich, da dieser automatisch eine Umwandlung von Gleitkommazahlen zu Integerzahlen und umgekehrt vornimmt. Die CREAL-Anweisung dient beim Interpreter dazu, die Kompatibilität zu compilierten Programmen zu gewährleisten.

Vergleichen Sie hierzu auch: CINT, INT, FIX, ROUND

DATA

Format: DATA ⟨Konstantenliste⟩

Zweck:

Übergeben von vorher bekannten Konstanten an das Programm.

Anwendung:

In DATA-Anweisungen werden numerische und nichtnumerische Konstanten im Programm fest vorgegeben, die mit der READ-Anweisung ein- oder mehrmals gelesen und an Variable während des Programmablaufs übergeben werden.

DATA-Anweisungen werden ohne zugehörige READ-Anweisung vom laufenden Programm übergangen, da sie selbst keine Funktion besitzen. Die im Programm vorhandenen DATA-Zeilen dürfen an jeder für die Programmübersicht sinnvollen Stelle eingefügt werden, d.h., die Programmausführung wird hierdurch nicht beeinträchtigt.

In einer DATA-Zeile können Konstanten nacheinander angegeben werden, indem sie durch Kommata getrennt werden. Die zugehörige READ-Anweisung liest diese Konstanten linear bzw. nacheinander. Zu beachten ist, daß die READ-Anweisung immer zuerst auf die niedrigste Zeilennummer und dann auf die nächsthöhere Zeilennummer mit einer DATA-Anweisung zu greift, wenn die vorgegebenen Konstanten in der ersten DATA-Zeile durch eine READ-Anweisung gelesen sind.

Beispiel:

```
10 MODE 1
20 READ A$,B$,C$,N,D$,E$,O
30 DATA SCHNEIDER,COMPUTER DIVISION
40 DATA DER REELLE,64,k COLOUR MICRO COMPUTER,CPC
50 DATA 464
60 PRINT A$
70 PRINT
80 PRINT B$:PRINT:PRINT C$
90 PRINT:PRINT TAB (4) N;D$
100 PRINT:PRINT E$;O
110 END
```

Ergebnis:

```
SCHNEIDER

COMPUTER DIVISION

DER REELLE

    64 k COLOUR MICRO COMPUTER

CPC 464
```

Vergleichen Sie hierzu auch: READ, RESTORE

DEF FN

Format: `DEF FN` ⟨Name⟩`[(`⟨Parameterliste⟩`)]=`⟨Ausdruck⟩

Zweck:

Definieren einer benutzereigenen Funktion und Übergabe an einen Variablennamen.

Anwendung:

Die Anweisung `DEF FN` erlaubt es, Funktionen zu definieren, die wie eine mathematische Standardfunktion im Programm aufgerufen werden können. Sie kann nicht im Kommandomodus benutzt werden, da sonst die Fehlermeldung "`Invalid direct command`" erscheint. Die `DEF FN`-Anweisung muß mindestens einmal vom Programm durchlaufen sein, bevor auf diese selbstdefinierte Funktion zugegriffen werden kann.

Für ⟨Name⟩ gelten dieselben Regeln wie für sonstige Variablennamen. Die Angaben für ⟨Name⟩ und für ⟨Ausdruck⟩ müssen vom selben Typ sein. Wird die selbstdefinierte Funktion aufgerufen, so muß die Art und Anzahl der Parameter mit den bei der Definition der Funktion festgelegten Parametern übereinstimmen.

Beispiel:

```
10 MODE 2
20 DEF FNT$(X,Y)=STR$(X+Y)+"%"
30 DEF FNC(ZAHL)=ZAHL/10
40 PRINT"Zum Abbruch des Programms bitte ESC-Taste
druecken"
50 PRINT
60 INPUT "Bitte geben Sie eine Zahl ein: ",ZAHL1
70 INPUT "Bitte geben Sie eine zweite Zahl ein: ",ZAHL2
80 PRINT
90 PRINT "FNT$(ZAHL1,ZAHL2)="FNT$(ZAHL1,ZAHL2)
100 PRINT "FNC(ZAHL1)="FNC(ZAHL1)
110 GOTO 50
```

Ergebnis:

```
Zum Abbruch des Programms bitte ESC-Taste druecken

Bitte geben Sie eine Zahl ein: 3
Bitte geben Sie eine zweite Zahl ein: 4

FNT$(ZAHL1,ZAHL2)= 7%
FNC(ZAHL1)= 0.3
```

Vergleichen Sie hierzu auch: DEFINT, DEFSTR, DEFREAL

Format: DEFINT ⟨Liste von Anfangsbuchstaben⟩

Zweck:

Bestimmen eines Anfangsbuchstaben zum Typ Integer.

Anwendung:

DEFINT definiert die Variablen, deren Anfangsbuchstaben in der ⟨Liste von Anfangsbuchstaben⟩ steht, zum Typ Integer. Nach einer Typdeklaration ist es nicht mehr notwendig, ein Prozentzeichen "%" nach dem Variablennamen anzugeben.

Eine Variable, die als Integer definiert wurde, enthält automatisch nur ganzzahlige Werte.

Wird die Definitionsanweisung in der Form:

```
DEFINT I-N,Z
```

angegeben, so bedeutet das Minuszeichen "-" gleich "bis" und erfaßt alle Variab lennamen von "I" bis "N", die dann als Integer-, also als ganze Zahlen behandelt werden. Außerdem sind alle Namen mit dem Anfangsbuchstaben "Z" Integerzahlen.

Beispiel:

```
10 MODE 1
20 DEFINT I
30 I=PI
40 PRINT I
50 K%=1.56789
60 PRINT K%
70 END
```

Ergebnis:

```
3 2
```

Vergleichen Sie hierzu auch: DEFSTR, DEFREAL, DEF FN

Format: `DEFSTR` ⟨Liste von Anfangsbuchstaben⟩

Zweck:

Bestimmen eines Anfangsbuchstabens zum Typ Zeichenkette.

Anwendung:

`DEFSTR` definiert die Variablen, deren Anfangsbuchstaben in der ⟨Liste von Anfangsbuchstaben⟩ steht, zum Typ String (Zeichenkette). Nach einer Typdeklaration ist es nicht mehr notwendig, ein Dollarzeichen "$" nach dem Variablennamen anzufügen.

Wird die Definitionsanweisung in der Form:

```
DEFSTR A-L,S
```

angegeben, so bedeutet das Minuszeichen "-" gleich "bis" und erfaßt alle Variablennamen von "A" bis "L", die dann als Stringvariablen behandelt werden. Außerdem sind noch alle Namen mit dem Anfangsbuchstaben "S" Stringvariablen.

Beispiel:

```
10 MODE 1
20 DEFSTR S
30 S="FAMILIENGESCHICHTEN"
40 A$=STRING$(19,CHR$(&5F))
50 PRINT S: PRINT A$
60 END
```

Ergebnis:

```
FAMILIENGESCHICHTEN
```

Vergleichen Sie hierzu auch: DEFREAL, DEFINT, DEF FN

Format: DEFREAL ⟨Liste von Anfangsbuchstaben⟩

Zweck:

Bestimmen eines Anfangsbuchstabens zum Typ Reell.

Anwendung:

DEFREAL definiert die Variablen, deren Anfangsbuchstaben in der ⟨Liste von Anfangsbuchstaben⟩ steht zum Typ Reell.

Die Angabe von Anfangsbuchstaben in der ⟨Liste von Anfangsbuchstaben⟩ kann sowohl in Klein- als auch in Großbuchstaben erfolgen.

Vergleichen Sie hierzu auch: DEFSTR, DEFINT, DEF FN

DEG

Format: DEG

Zweck:

Umschalten der Berechnung auf Grad.

Anwendung:

DEG bewirkt, daß bei Aufruf einer mathematischen Funktion die Berechnung in Grad ausgeführt wird. Die Anweisung RAD setzt wieder auf die Berechnung im Bogenmaß (Radiant) zurück.

Vergleichen Sie hierzu auch: RAD, SIN, COS, TAN, ATN

Format: DELETE [⟨Zeilennummer1⟩] [-⟨Zeilennummer2⟩]

Zweck:

Löschen von Zeilen und Zeilenbereichen.

Anwendung:

Die DELETE-Anweisung dient dem Löschen von Programmzeilen, wobei diese vorhanden sein müssen. Ohne der beiden Angaben ⟨Zeilennummer1⟩ und ⟨Zeilennummer2⟩ löscht die DELETE-Anweisung alle Programmzeilen. Wird nur ⟨Zeilennummer1⟩ angegeben, so wird die entsprechende Zeile mit der Zeilennummer ⟨Zeilennummer1⟩ gelöscht. Mit der zusätzlichen Angabe von ⟨Zeilennummer2⟩ kann ein Zeilennummernbereich gelöscht werden.

Beispiele:

DELETE löscht alle Zeilen

DELETE 30 löscht die Zeile Nr. 30

DELETE 65-100 löscht die Zeilen von Nr. 65- Nr. 100 einschließlich

DELETE -100 löscht alle Zeilen bis Nr. 100 einschließlich

DELETE 70- löscht alle Zeilen ab Nr. 70 einschließlich

Vergleichen Sie hierzu auch: EDIT, RENUM

DI

Format: DI

Zweck:

Verzögern von Unterprogrammsprüngen.

Anwendung:

Durch die EVERY-Anweisung werden in bestimmten Zeitintervallen Sprünge zu Unterprogrammen ausgeführt. Unterprogrammsprünge können in bestimmten Fällen unerwünscht sein, wenn sie den Programmablauf in Frage stellen. Die DI-Anweisung verschiebt diese Sprünge zu den Unterprogrammen, bis eine EI-Anweisung erreicht wird.

Erreicht BASIC die EI-Anweisung, so werden die nicht ausgeführten Unterprogrammsprünge nachgeholt, bevor das Hauptprogramm fortgesetzt wird, d.h., BASIC merkt sich die Anzahl der verzögerten bzw. verhin derten Unterprogrammsprünge.

Vergleichen Sie hierzu auch: EI, EVERY, AFTER, REMAIN

Format: `DIM` ⟨Feldvariable⟩`(X)`

bzw.

`DIM` ⟨Feldvariable⟩`(A,B,C,...,L)`

Zweck:

Definieren der Elementeanzahl eines Feldes und Reservieren von Speicherplatz.

Anwendung:

Die `DIM`-Anweisung stellt für Feldvariablen die benötigten Speicher plätze zur Verfügung. Für jeden Index steht der maximal zulässige Wert in der Liste der Feldvariablen.

Es sind maximal 12 Indizes von (A ... bis L) in der Praxis möglich. Die Anzahl wird hierbei durch den freien Platz im Arbeitsspeicher beschränkt.

In einer Anweisung können alle Feldvariablen stehen. Diese Anweisung muß im Programm vor dem ersten Zugriff auf das entsprechende Feld liegen, da sonst der Speicherplatz nicht von BASIC reserviert ist und die Fehlermeldung "Subscript out of range in ⟨Zeilennummer⟩" ausgegeben wird. Der niedrigste erlaubte Index ist 0. Werden die Indizes im Programm nicht größer als 10, so ist die `DIM`-Anweisung für das entsprechende Feld entbehrlich.

Überschreitet die Feldgröße den freien Speicherplatz, so erscheint die Fehlermeldung `"Memory full in` ⟨Zeilennummer⟩`"` auf dem Monitor. Die Fehlermeldung `"Array already dimensioned"` wird ausgegeben, wenn der Name der Feldvariablen bereits existiert.

1. Beispiel:

```
5 REM **** PROGRAMM BRICHT MIT FEHLERMELDUNG AB ****
10 MODE 1
20 FOR I=0 TO 30
30 BUCHSTABEN$(I)=CHR$(32+I)
40 PRINT I;
50 NEXT I
60 END
```

Ergebnis:

```
0  1  2  3  4  5  6  7  8  9  10
Subscript out of range in 30
Ready
```

Die Fehlermeldung erscheint, da mit dem Programm auf ein Feldelement größer 10 zugegriffen wurde, ohne daß ein Feld dieser Größe mit einer `DIM`-Anweisung dimensioniert wurde.

2. Beispiel:

```
5  REM **** PROGRAMM OHNE FEHLERMELDUNG ****
10 MODE 1
20 ANZAHL=25
30 DIM ZEICHEN$(ANZAHL)
40 FOR I=0 TO ANZAHL
50 ZEICHEN$(I)=CHR$(65+I)
60 NEXT I
70 FOR K=0 TO ANZAHL
80 PRINT ZEICHEN$(K);
90 NEXT K
100 END
```

Ergebnis:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Vergleichen Sie hierzu auch: ERASE

DRAW

Format: `DRAW` ⟨X-Koordinate⟩`,`⟨Y-Koordinate⟩`[,`⟨Code⟩`]`

Zweck:

Zeichnen von Linien ab der Graphik-Cursorposition zu einer absoluten Position.

Anwendung:

Die DRAW-Anweisung zeichnet eine Linie von der aktuellen Graphik-Cursorposition zu dem durch X- und Y-Koordinate bestimmten Punkt. Dieser Punkt wird nach dem Zeichnen zur neuen Graphik-Cursorposition. Die Werte für die X- und Y-Koordinaten beziehen sich auf die Pixel-Koordinaten. Das Bildschirmfenster besteht aus 640 horizontalen und 400 vertikalen Bildpunkten, den sogenannten Pixels. Die untere linke Ecke des Bildschirmfensters besitzt die Koordinaten 0,0, die rechte obere Ecke die Koordinaten 639,399. Das `DRAW`-Kommando wird nur teilweise von der Formatanweisung `MODE` beeinflußt, d.h., die eingegebenen Koordinaten behalten ihre Gültigkeit, nur die Liniendicke bzw. Pixelgröße und die Farbauswahl werden verändert.

Beispiel:

```
10 FOR I=0 TO 2
20 BORDER 18
30 MODE I
40 LOCATE 1,13
50 PRINT "MODE";I
60 PLOT 0,0
70 DRAW 640,400,14
80 PLOT 640,0
90 DRAW 0,400,15
100 FOR K=2000 TO 1 STEP -1:NEXT K: ***ZEITSCHLEIFE***
110 NEXT I
120 END
```

Code-Tabelle

<Code>		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20.	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1><24	20	1
15*	= Farbcode	16><11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "><".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

`DRAWR, PLOT, PLOTR, MOVE, MOVER, ORIGIN, TEST, TESTR, XPOS, YPOS`

Format: DRAWR ⟨rel.X-Koordinate⟩,⟨rel.Y-Koordinate⟩[,⟨Code⟩]

Zweck:

Zeichnen einer Linie relativ zur Graphik-Cursorposition.

Anwendung:

Die DRAWR-Anweisung zeichnet relativ zur aktuellen Graphik-Cursorposition eine Linie. Der Startpunkt der Linie wird bestimmt durch die aktuelle Graphik-Cursorposition. Die X-Koordinate des Zielpunktes wird durch die X-Koordinate der aktuellen Graphik-Cursorposition + ⟨rel. X- Koordinate⟩ bestimmt. Die Y-Koordinate berechnet sich entsprechend der X-Koordinate. Das DRAWR-Kommando wird nur teilweise von der Formatan weisung MODE beeinflußt, d.h., die eingegebenen Koordinaten behalten ihre Gültigkeit, nur die Liniendicke bzw. die Pixelgröße und die Farbauswahl werden verändert.

Beispiel:

```
10 MODE 1
15 BORDER 18
20 ORIGIN 450,150
30 FOR I=1 TO 8
40 READ X,Y
50 DATA 0, 100, -100, 100, -100, 0, -100, -100, 0, -100,
100, -100, 100, 0, 100, 100
60 DRAWR X,Y,7
70 NEXT I
80 GOTO 80
```

Ergebnis:

Dieses Programm zeichnet ein Achteck. Zeile 80 unterdrückt die "Ready"-Meldung. Das Programm muß mit zweimaligem Drücken der ESC- Taste abgebrochen werden.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

DRAW, PLOT, PLOTR, MOVE, MOVER, ORIGIN, TEST, TESTR, XPOS, YPOS

EDIT

Format: `EDIT <Zeilennummer>`

Zweck:

Editieren von bestimmten Programmzeilen.

Anwendung:

Sie geben die Anweisung `EDIT` und die Zeilennummer derjenigen Zeile im Kommandomodus ein, in der Sie Veränderungen vornehmen wollen. Die Zeile steht dann mit ihrem vollen Inhalt auf dem Monitor.

Beispiel:

`EDIT 185` Zeile 185 erscheint auf dem Bildschirmfenster

Vergleichen Sie hierzu auch: `LIST, RENUM, DELETE`

Format: EI

Zweck:

Abschalten der Verzögerung von Unterprogrammsprüngen.

Anwendung:

Durch die EVERY-Anweisung werden in bestimmten Zeitintervallen Sprünge zu Unterprogrammen ausgeführt. Diese können in bestimmten Fällen unerwünscht sein.

Die DI-Anweisung verschiebt diese Sprünge zu den Unterprogrammen, bis eine EI-Anweisung erreicht wird.

Erreicht BASIC die EI-Anweisung, so werden die nicht ausgeführten Unterprogrammsprünge nachgeholt, bevor das Hauptprogramm fortgesetzt wird. BASIC merkt sich also die Anzahl der verzögerten bzw. verhinderten Unterprogrammsprünge und holt sie alle nach. Aufgeholt werden die verzögerten Programmsprünge dann nicht, wenn mit zweimaligem Drücken der ESC-Taste das Programm abgebrochen wurde.

Sind Unterprogrammsprünge nachzuholen, so werden zunächst die des Timers mit der höchsten Priorität berücksichtigt.

Vergleichen Sie hierzu auch: DI, EVERY, AFTER, REMAIN

END

Format: END

Zweck:

Beenden der Programmausführung.

Anwendung:

Die END-Anweisung beendet die Programmausführung, schließt alle eröffneten Kassetten-FILES und kehrt in den Kommandomodus zurück. Folgt auf END keine weitere ausführbare Anweisung, so darf END auch fehlen. Bei BASIC-Programmtests ist es sinnvoll, innerhalb eines Programmes END- Anweisungen anzulegen, um z.B. Fehlerquellen zu bestimmen. Die Anweisung kann mit dem CONT-Kommando aufgehoben werden.

Beispiel:

```
10 MODE 1
20 PRINT "Vor der 'END-Anweisung'"
30 END
40 PRINT "Nach der 'END-Anweisung'"
```

Ergebnis:

```
Vor der 'END-Anweisung'
Ready
```

Hinweis:

Geben Sie nun `CONT` im Direktmodus ein, damit das Programm fortgesetzt wird.

```
CONT
Nach der 'END'-Anweisung
Ready
```

Vergleichen Sie hierzu auch: `CONT`, `*Break*`, `STOP`

ENT

Format:

ENT ⟨Hüllkurvennummer⟩,⟨Schrittzahl⟩,⟨Schrittgröße⟩,
⟨Pausenlänge⟩ [,...] [,...] [,...]

Zweck:

Definieren einer Tonhüllkurve.

Anwendung:

Die ENT-Anweisung kann benutzt werden, um eine Tonhüllkurve zu definieren. Die Tonhüllkurve beinflußt nicht die Tonlänge bzw. Dauer. Es werden hiermit kleine Frequenzverschiebungen bewirkt, die der gespielten Note eine bestimmte Charakteristik verleihen. Vergleichbar ist dieser Effekt mit einem Vibrato.

Die nach der ENT-Anweisung stehenden Variablen besitzen folgende Bedeutung:

⟨Hüllkurvennummer⟩:

Sie ordnet der Hüllkurve einen bestimmten Wert zu, damit dieser, eingesetzt bei der SOUND-Anweisung, zum Aufruf der definierten Tonhüllkurve genutzt werden kann.

⟨Schrittzahl⟩:

Liegt im Bereich von 0 bis 239 und definiert die schrittweise Veränderung relativ zur momentanen Tonperiode.

⟨Schrittgröße⟩

Der Wertebereich liegt zwischen -128 und 127 und definiert die Differenz zwischen den jeweiligen Tonperioden.

⟨Pausenlänge⟩:

Liegt im Bereich von 0 bis 255 und definiert die Wartezeit zwischen den Frequenzänderungen. Eine ⟨Pausenlängen⟩-Einheit ist gleich 0,01 Sekunden.

Die ENT-Anweisung erlaubt es, bis zu fünf unterschiedliche Frequenzänderungsverläufe anzugeben, so daß die im Aufruf dargestellte Form um weitere 4 gleichwertige Angaben erweitert werden kann. Sollte die Notenlänge bzw. Dauer kürzer als die Zeitbasis bei der ENT-Anweisung sein, so werden die Restanweisungen nicht mehr ausgeführt.

Vergleichen Sie hierzu auch:

ENV, SOUND, RELEASE, SQ, ON SQ GOSUB

ENV

Format:

ENV ⟨Hüllkurvennummer⟩, ⟨Schrittzahl⟩, ⟨Schrittgröße⟩, ⟨Pausenlänge⟩ [,...] [,..] [,...]

Zweck:

Definieren einer Lautstärkenhüllkurve.

Anwendung:

Die ENV-Anweisung kann benutzt werden, um eine Lautstärkenhüllkurve zu definieren. Diese beinflußt die Tonlänge bzw. Tondauer. Hiermit können Lautstärkenverschiebungen vorgenommen und der Zeitintervall der Verschiebung bestimmt werden.

Hiermit kann den gespielten Noten eine bestimmte Musikgerät abhängige Charakteristik verliehen werden, z.B. bei einem Klavier der Anschlag einer Taste und das Verklingen des Tones.

Die nach der ENT-Anweisung stehenden Variablen besitzen folgende Bedeutung:

⟨Hüllkurvennummer⟩:

Sie ordnet der Hüllkurve einen bestimmten Wert zu, damit dieser, eingesetzt bei der SOUND-Anweisung, zum Aufruf der definierten Lautstärkenhüllkurve genutzt werden kann. Der Wertebereich liegt zwischen 1 und 15 und muß angegeben werden.

⟨Schrittzahl⟩:

Liegt im Bereich von 0 bis 127 und definiert die schrittweise Veränderung relativ zur momentanen Lautstärke.

⟨Schrittgröße⟩

Der Wertebereich liegt zwischen -128 und 127 und definiert die Differenz zwischen den jeweiligen Lautstärkeveränderungen.

⟨Pausenlänge⟩:

Liegt im Bereich von 0 bis 125 und definiert die Wartezeit zwischen den Lautstärkeänderungen. Eine ⟨Pausenlängen⟩-Einheit ist gleich 0,01 Sekunden.

Die ENV-Anweisung erlaubt es, die Veränderung der Lautstärke in bis zu fünf unterschiedlichen Abschnitten anzugeben, so daß die im Aufruf dargestellte Form um weitere 4 gleichartige Angaben erweitert werden kann. Sollte die errechenbare Gesamtdauer aller Abschnitte kürzer als die Länge der Note sein, so wird die Notenlänge gekürzt.

Vergleichen Sie hierzu auch:

ENT, SOUND, RELEASE, SQ, ON SQ GOSUB

EOF

Format: `EOF`

Zweck:

Erkennen des Dateiendes bei einem sequentiellen Lesen von Kassette.

Anwendung:

Mit `EOF` wird das Dateiende erkannt. Wurde das Dateiende erreicht, so ergibt diese Anweisung den Wert -1 (wahr).

`EOF` muß vor der `INPUT-#9`-Anweisung abgefragt werden, da sonst bei Dateiende die Fehlermeldung "`EOF met in` <Zeilennummer>" ausgegeben wird und ein Programmabbruch erfolgt.

Ein Anwendungsbeispiel finden Sie bei der OPENOUT-Anweisung.

Vergleichen Sie hierzu auch:

`OPENOUT, CLOSEOUT, OPENIN, CLOSEIN, PRINT, INPUT, LINE INPUT, WRITE`

Format: ERASE

Zweck:

Löschen der von DIM definierten Feldvariablen.

Anwendung:

Die ERASE-Anweisung gibt vom Programm nicht mehr benötigte Speicherbereiche frei, indem einzelne mit der DIM-Anweisung dimensionierte Felder gelöscht werden. Nun ist es möglich, eine bereits dimensionierte Variable neu zu dimensionieren.

Beispiel:

```
10 MODE 2
20 ON ERROR GOTO 80
30 DIM WERT(12)
40 WERT(I)=I
50 PRINT WERT(I);
60 I=I+1:GOTO 40
80 PRINT CHR$(10)CHR$(10)
90 PRINT "Vorsicht, das Feld ist mit ";I-1;"zu klein
dimensioniert!"
100 PRINT
110 INPUT "Wie groß soll das Feld dimensioniert
werden ";ANZAHL
120 REM ****** ERASE - ZEILE s. u. *****
130 DIM WERT(ANZAHL)
140 RESUME
```

Ergebnis:

```
0 1 2 3 4 5 6 7 8 9 10 11 12

Vorsicht, das Feld ist mit 12 zu klein dimensioniert!

Wie groß soll das Feld dimensioniert werden? 20
Array already dimensioned in 130
Ready
```

Die Fehlermeldung erscheint, weil das Feld `WERT` bereits dimensioniert war.

Dieser Fehler tritt nicht auf, wenn Sie in dem Programm die Zeile 120 abändern in:

```
120 ERASE WERT
```

Zeile 120 bewirkt eine Freigabe zur Neudimensionierung des Feldes `WERT` und löscht dabei den gesamten Feldinhalt.

Vergleichen Sie hierzu auch: `DIM`

Format: ERL

Zweck:

Bestimmen der Fehlerzeile.

Anwendung:

Die Funktion ERL wird in der Regel in Fehlerbehandlungsroutinen verwendet und bestimmt die Zeilennummer, in der ein Fehler aufgetreten ist. Ist kein Fehler vorhanden, so wird der Wert null "0" ausgegeben.

Vergleichen Sie hierzu auch: ERR, ON ERROR, ERROR und das Kapitel "Fehlermeldungen und Fehlercodes".

ERR

Format: `ERR`

Zweck:

Bestimmen des Fehlercodes.

Anwendung:

Die Funktion `ERR` wird in der Regel in Fehlerbehandlungsroutinen angewendet und bestimmt den Fehlercode des Fehlers, der aufgetreten ist. Ist kein Fehler vorhanden, so wird der Wert null "0" ausgegeben.

Beispiel:

```
10 GOTO 9999
```

```
RUN
```

Ergebnis:

Es wird die Fehlermeldung "`Line does not exist in 10`" ausgegeben. Nach Eingabe von:

```
?err
```

wird der dieser Fehlermeldung entsprechende Fehlercode: `8` ausgegeben.

Vergleichen Sie hierzu auch: `ERL`, `ON ERROR`, `ERROR` und das Kapitel "Fehlermeldungen und Fehlercodes".

ERROR

Format: ERROR ⟨Fehlercode⟩

Zweck:

Dient der Fehlersimulation.

Anwendung:

Mit ERROR kann durch BASIC eine Fehlermeldung ausgegeben werden, die der Fehlermeldung entspricht, die bei einem entsprechenden Fehler auftreten würde. Der ⟨Fehlercode⟩ muß zwischen 1 und 255 liegen. ERROR kann angewendet werden, um im Programm einen Fehler zu erzeugen, der dann durch eine Fehlerbehandlungsroutine ausgewertet wird. ⟨Fehler codes⟩ größer als 30 lassen die Meldung "Unknown error in ⟨Zeilennummer⟩" erscheinen.

Vergleichen Sie hierzu auch: ON ERROR GOTO, ERR, ERL

EVERY

Format: EVERY ⟨Zeitangabe⟩,[⟨Timer⟩] GOSUB ⟨Zeilennummer⟩

Zweck:

Wiederholtes Aufrufen von einem Unterprogramm in vorgegebenen Zeit- Intervallen.

Anwendung:

Mit der EVERY-Anweisung können Unterprogramme, vorprogrammiert durch den Wert ⟨Zeitangabe⟩, in regelmäßigen Zeitintervallen automatisch aufgerufen werden. ⟨Zeitangabe⟩=1 entspricht 0,02 Sekunden. Nach Ablauf der Zeitangabe wird das Unterprogramm mit GOSUB ⟨Zeilennummer⟩ aufgerufen und der Zähler für den nächsten Programmdurchlauf wieder auf Null gesetzt.

⟨Timer⟩ ist einer der vier Zeitgeber, anwählbar mit den Werten 0 bis 3. Wird die Angabe ⟨Timer⟩ nicht genutzt, dann ist Timer 0 gesetzt. Die Zeitgeber besitzen unterschiedliche Prioritäten. ⟨Timer⟩ gleich 3 besitzt die höchste, ⟨Timer⟩ gleich 0 die niedrigste Priorität.

Die ⟨Zeilennummer⟩ gibt die Anfangszeile eines Unterprogramms an, das mit RETURN beendet werden muß.

Die bei der EVERY-Anweisung benutzten ⟨Timer⟩ sind identisch mit denen der AFTER-Anweisung. Dieses ist zu beachten, da durch eine nachfolgende AFTER-Anweisung der angegebene ⟨Timer⟩ überschrieben wird. Dieses gilt auch für die umgekehrte Anweisungsfolge.

Beispiel:

```
10 EVERY 100 GOSUB 60
20 EVERY 10,1 GOSUB 110
25 EVERY 5,3 GOSUB 200
30 PRINT "*";
40 GOSUB 130
50 GOTO 30
60 DI
70 PRINT " 100 "
80 GOSUB 130
90 EI
100 RETURN
110 PRINT "2";
120 RETURN
130 FOR I=0 TO 1000:NEXT I:RETURN
200 PRINT "#";:RETURN
```

Vergleichen Sie hierzu auch: AFTER, REMAIN, EI, DI

EXP

Format: `EXP(`⟨numerischer Ausdruck⟩`)`

Zweck:

Berechnet Exponentialfunktionen.

Anwendung:

Die Exponentialfunktion `EXP` berechnet den Wert "e" hoch ⟨numerischer Ausdruck⟩. Die Exponentialfunktion als Umkehrfunktion von `LOG` liefert für `EXP(1)` den Wert 2.71828183, die Eulersche Zahl "e". Der natürliche Logarithmus der Eulerschen Zahl ist also 1.

Der ⟨numerische Ausdruck⟩ sollte kleiner als 89 sein. Wird der Wert 89 für ⟨numerischer Ausdruck⟩ überschritten, erscheint die Fehlermeldung `Overflow`. Danach wird als Ergebnis die größte darstellbare Zahl angenommen und das Programm fortgesetzt.

Beispiel:

```
Print EXP(6.876)
```

Ergebnis:

```
968.743625
```

Vergleichen Sie hierzu auch: `LOG, LOG10`

Format: FIX(⟨numerischer Ausdruck⟩)

Zweck:

Umwandeln eines numerischen Ausdrucks in einen ganzzahligen Wert.

Anwendung:

FIX(⟨numerischer Ausdruck⟩) ergibt den ganzzahligen Wert vom vorgegebenen Ausdruck. Es werden alle Nachkommastellen entfernt. Mit B=ABS(A-FIX(A)) können Sie den gebrochenen Anteil von A ermitteln, d.h. die Zahlen rechts vom Dezimalpunkt.

Beispiel:

```
? FIX(1.7356),FIX(-1.8)
```

Ergebnis:

```
1          -1
```

Vergleichen Sie hierzu auch: CINT, INT, ROUND, CREAL

FOR ... NEXT

Format:

FOR ⟨Variable⟩=⟨Startwert⟩ TO ⟨Endwert⟩[STEP ⟨Schrittweite⟩]

.
⟨Anweisungen⟩
⟨Anweisungen⟩
.
NEXT ⟨Variable⟩

Zweck:

Wiederholte Ausführung eines bestimmten Programmteils.

Anwendung:

Die FOR-NEXT-Anweisung erlaubt es, in einem Programm Programmschleifen zu realisieren, d.h., ein bestimmter Programmteil zwischen FOR und NEXT wird definiert häufig wiederholt.

FOR ist die Laufanweisung, NEXT bestimmt das Schleifenende.

In der Laufanweisung werden die Schleifendurchgänge ab dem ⟨Startwert⟩ gezählt und der Wert der ⟨Variablen⟩ übergeben. Ist der vorgegebene ⟨Endwert⟩ noch nicht erreicht, so werden die nach der Laufanweisung stehenden ⟨Anweisungen⟩ durchlaufen.

Erreicht der Wert der ⟨Variablen⟩ den ⟨Endwert⟩, so wird die nächste Zeile bzw. ⟨Anweisung⟩ nach NEXT vom Programm abgearbeitet.

Werden die ⟨Schrittweite⟩ und STEP nicht angegeben, so wird für die ⟨Schrittweite⟩ der Wert "1" angenommen.

Beispiel 1:

```
90 MODE 1
100 FOR K=32 TO 255
110 PRINT K;" = ";CHR$K)
, 120 NEXT K
130 PRINT "Dies ist der Zeichensatz des CPC 464"
140 PRINT "mit dem entsprechenden ASCII-Wert"
145 PRINT:PRINT
150 LOCATE 1,24:PRINT "Programmabbruch mit ESC-
Taste!"
160 GOTO 160
```

Werden STEP und die ⟨Schrittweite⟩ angegeben, so berechnen sich die Schleifendurchgänge nach diesem Wert. Außerdem kann man negative Werte für die ⟨Schrittweite⟩ angeben und damit rückwärts zählen.

Beispiel 2:

```
90 MODE 1
100 FOR J=255 TO 32 STEP -2
110 PRINT J;" = ";CHR$J),
120 NEXT J:PRINT
130 PRINT "Vom Zeichensatz des CPC 464"
140 PRINT "ist nur jedes zweite Zeichen ausgegeben wor-
den!"
150 PRINT:PRINT
160 LOCATE 1,24: PRINT "Programmabbruch mit ESC-
Taste!"
170 GOTO 170
```

Vergleichen Sie hierzu auch: WHILE WEND

FRE

Format: `FRE(0)` bzw. `PRINT FRE(n"")`

Zweck:

Ermitteln des noch frei verfügbaren Speicherplatzes.

Anwendung:

`PRINT FRE(0)` ermittelt die Größe des noch nicht belegten, also frei verfügbaren Arbeitsspeichers unter BASIC und gibt sie auf dem Monitor aus.

Die Form `PRINT FRE("")` leitet eine "garbage collection" ein und ermittelt danach den noch verfügbaren freien Speicherplatz. Eine garbage collection löscht alle nicht mehr benötigten Stringvariablen und vergrößert damit den vorhandenen freien Speicherplatz.

Der ermittelte freie Speicherplatz kann auch innerhalb eines Programms an eine Variable übergeben werden.

Beispiel:

```
10 DIM FELD(5000)
20 MODE 2
30 A=FRE(0)
40 PRINT "Der freie Speicherplatz betraegt ";A;"
Bytes"
50 END
```

Ergebnis:

```
Der freie Speicherplatz betraegt 18405 Bytes
```

Format: GOSUB ⟨Zeilennummer⟩

Zweck:

Verzweigen in ein Unterprogramm.

Anwendung:

Die GOSUB-Anweisung verzweigt in ein mit der Zeilennummer ⟨Zeilennummer⟩ beginnendes Unterprogramm. Unterprogramme werden in einem BASIC- Programm dann eingearbeitet, wenn die gleiche Folge von Anweisungen mehr als einmal an unterschiedlichen Stellen des Programms durchlaufen werden müssen.

Die Zeilennummer muß immer existieren, da sonst die Fehlermeldung "Line does not exist in ⟨Zeilennummer⟩" ausgegeben wird. Unterprogramme, die mit GOSUB angesprungen werden, müssen mit der Anweisung RETURN abgeschlossen sein. RETURN beendet die Anweisungs folge im Unterprogramm und bewirkt einen Rücksprung zu der unmittelbar auf GOSUB folgenden Anweisung im Hauptprogramm. Hinter der RETURN-Anweisung steht keine Zeilennummer, so daß von verschiedenen Hauptprogrammstellen in das Unterprogramm gesprungen werden kann und auch der entsprechende Rücksprung erfolgen kann.

Beispiel:

```
10 REM *** HAUPTPROGRAMM ***
20 MODE 1
30 I=1
40 A$="wenn der Vater mit dem Sohne"
50 GOSUB 90
60 A$="auf die Fasnacht geht ..."
70 GOSUB 90
80 END
90 REM *** UNTERPROGRAMM ***
100 PRINT CHR$(10)STR$I);". ";
110 A$=UPPER$(A$)
120 PRINT A$
130 I=I+1
140 RETURN
```

Ergebnis:

```
1. WENN DER VATER MIT DEM SOHNE
2. AUF DIE FASNACHT GEHT ...
```

Das in Zeile 80 eingefügte END-Kommando ist notwendig, damit das Unterprogramm nicht ohne GOSUB durchlaufen wird. Sonst würde die Fehlermeldung "Unexpected RETURN in ⟨Zeilennummer⟩" erscheinen.

Vergleichen Sie hierzu auch:

RETURN, ON GOSUB, ON ERROR GOSUB, CALL

GOTO

Format: GOTO ⟨Zeilennummer⟩

Zweck:

Unbedingtes Verzweigen zu einer bestimmten Programmzeile.

Anwendung:

GOTO leitet einen unbedingten Sprung ein, d.h. es wird ein Programmsprung ohne weitere Bedingung in die Programmzeile der ⟨Zeilennummer⟩ ausgeführt. Diese in der GOTO-Anweisung angegebene Zeilennummer muß vorhanden sein, da sonst die Fehlermeldung `Line does not exist in` ⟨Zeilennummer⟩ erscheint. Dieses ist eine häufig vorkommende Fehler meldung, wenn unkontrolliert Zeilen gelöscht werden.

Diese Anweisung ist auch im Kommandomodus sehr wertvoll, um ein abgebrochenes Programm ab einer bestimmten Zeilennummer neu zu starten, ohne hierbei die Variablen zu löschen, wie es z.B. mit der RUN-Anweisung geschieht.

Beispiel:

```
10 MODE 0
20 PRINT CHR$; TIME;
30 GOTO 20
```

Ergebnis:

Dieses kleine Programm gibt laufend die vergangene Zeit ab dem Einschalten des ⟩⟩CPC 464 ⟨⟨ in 1/300 Sekunden aus.

Vergleichen Sie hierzu auch:

RUN, GOSUB, ON GOTO, ON ERROR GOTO, ON GOSUB

HEX$

Format: `HEX$ (⟨numerischer Ausdruck⟩)`

Zweck:

Bestimmen des hexadezimalen Werts.

Anwendung:

Die `HEX$`-Anweisung ergibt eine Zeichenkette, die den hexadezimalen Wert des ⟨numerischen Ausdrucks⟩ darstellt. Vor der Umwandlung wird der ⟨numerische Ausdruck⟩ gerundet und in eine Integerzahl umgewandelt.

Beispiel:

```
PRINT HEX$(74),HEX$(-100)
```

Ergebnis:

```
4A                      FF9C
```

Vergleichen Sie hierzu auch: `BIN$, STR$`

Format: `HIMEM`

Zweck:

Ermitteln der höchsten von BASIC benutzten Adresse.

Anwendung:

Die `HIMEM`-Anweisung liefert die höchste Adresse, die BASIC benutzt. Verwendet wird es, wenn man den für BASIC freien Speicherplatz mit `MEMORY` ändern will. Wurde z.B. mit `ADR=HIMEM` vorher die höchste Adresse abgefragt und in die Variable `ADR` übergeben, so kann nach der Änderung diese leicht wieder rückgängig gemacht werden mit `MEMORY=ADR`.

Vergleichen Sie hierzu auch: `MEMORY, FRE`

IF ... THEN

Format:

IF ⟨Bedingung⟩ THEN ⟨Anweisung⟩ [ELSE ⟨Anweisung⟩]

bzw.

IF ⟨Bedingung⟩GOTO ⟨Zeilennummer⟩ [ELSE⟨Anweisung ⟩]

Zweck:

Vorgegebene Bedingung(en) abfragen und bedingte Programmsprünge ausführen.

Anwendung:

Mit einer IF-Anweisung können ⟨Bedingungen⟩, die in einem Programm variabel auftreten, abgefragt und für bedingte Anweisungen in dem Programm genutzt werden.

Mit der IF-Anweisung wird der logische Ausdruck, der in der Bedingung enthalten ist, ausgewertet. Ist dieser "wahr", so wird die erste Anweisung ausgeführt. Wurde ELSE mit mindestens einer Anweisung angegeben, so wird diese ausgeführt, wenn der logische Ausdruck "falsch" ist. Fehlt ELSE mit nachgestellter Anweisung, so wird im Programm mit der nächsten Zeile fortgefahren.

Beispiel:

```
5  REM ***** IF ... THEN ... ELSE ... TEST *****
10 MODE 2
20 PRINT "Bitte geben Sie ein!"
22 PRINT "Einmal zwei gleiche Zahlen und"
24 PRINT "einmal zwei verschiedene Zahlen!":PRINT
30 GOSUB 1010
40 IF ZAHL1=ZAHL2 THEN GOTO 70
50 PRINT"Die erste Zahl ist nicht gleich der zweiten
Zahl."
60 GOTO 80
70 PRINT"Die erste Zahl ist gleich der zweiten Zahl."
75 REM ***** ERSTE EINGABE BEENDET ********
80 GOSUB 1010
90 IF ZAHL1=ZAHL2 THEN PRINT"Zahl1 ist gleich Zahl2."
ELSE PRINT "Zahl1 ist nicht gleich Zahl2."
100 PRINT:PRINT:END
1000 REM ***** UNTERPROGRAMM ZUR ZAHLENEINGABE
******
1010 PRINT: INPUT"Die erste Zahl ";ZAHL1
1020 INPUT"Die zweite Zahl";ZAHL2
1030 RETURN
```

Programmzeile 40:

```
40 IF ZAHL1=ZAHL2 THEN GOTO 70
```

Hier wird ein bedingter Sprung zur Zeile 70 ausgeführt, wenn beide Zahlen gleich sind. Sind diese verschieden, so wird der Programmablauf mit Zeile 50 fortgesetzt.

Hinweis:

In Zeile 40 kann statt `THEN GOTO` auch nur `GOTO` oder nur `THEN` geschrieben werden, d.h. diese Kurzanweisungen entsprechen in ihrer Wirkung der in Zeile 40 verwendeten.

Programmzeile 90:

```
90 IF ZAHL1=ZAHL2 THEN PRINT"Zahl1 ist gleich Zahl2."
ELSE PRINT "Zahl1 ist nicht gleich Zahl2."
```

Hier wird kein bedingter Sprung, sondern eine bedingte Anweisung ausgeführt. Ist `ZAHL1=ZAHL2`, so ist die Bedingung "wahr" und die erste `PRINT`-Anweisung wird ausgeführt. Ist sie "falsch", d.h. die beiden Zahlen sind ungleich, wird die nach `ELSE` folgende Anweisung abgearbeitet. Die folgende Anweisung steht in beiden Fällen in Zeile 100.

Vergleichen Sie hierzu auch: `OR, AND, NOT, WHILE ... WEND`

Format: INK ⟨Code,Farbcode⟩[,⟨Farbcode⟩]

Zweck:

Farbaufruf für PAPER (Hintergrundfarbe) bzw. PEN (Zeichenfarbe) und Bestimmung eines Farbwechsels.

Anwendung:

Die INK-Anweisung besitzt mindestens zwei Codezahlen. Sie werden gebraucht, um die Farbe von PAPER bzw. PEN zu wechseln. Die erste Codezahl bezeichnet den bei PAPER bzw. PEN verwendeten Code. Der an zweiter Stelle anzugebende Farbcode bestimmt die neue Farbe, die PEN bzw. PAPER annimmt.

Nach dem Einschalten des ⟩⟩ CPC 464 ⟨⟨ besitzt PAPER den Code 0, PEN den Code 1 und es wird MODE 1 vorgegeben, d.h die Farbe von PAPER ist laut Farbcodetabelle blau, die Farbe von PEN ist laut Farbcodetabelle hellgelb.

Beispiel für PAPER:

Wollen Sie nun die Farbe von PAPER im MODE 1 von blau nach grün wechseln, so ist dies über den Code von der PAPER-Anweisung nicht direkt möglich. Sie geben folgenden Befehl ein:

```
INK 0,9
```

Ergebnis:

Das Bildschirmfenster erscheint nach Eingabe von RETURN grün.

Beispiel für PEN:

Wenn Sie die Farbe für PEN in hellweiß wechseln wollen, so ist dies in MODE 1 über den Code der PEN-Anweisung nicht direkt möglich. Geben Sie hierzu ein:

```
INK 1,26
```

Ergebnis:

Die Farbe der Zeichen ist nun hellweiß und der Hintergrund grün.

Die INK-Anweisung ermöglicht es weiterhin, die Hintergrundfarbe und die Zeichenfarbe zwischen zwei Farben wechseln zu lassen. Dieses wird durch die Angabe des zweiten Farbcodes ermöglicht. Die Farben wechseln dann zwischen der ersten und der zweiten Farbe.

Beispiel:

```
10 MODE 1
20 BORDER 18
30 PRINT:PRINT:PRINT
40 INK 1,3,26
50 PRINT" Programmabbruch mit ESC-Taste!"
60 GOTO 60
```

Ergebnis:

Die Zeichen blinken zwischen den Farben weiß und hellweiß.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

PEN, PAPER, BORDER, TEST, TESTR, SPEED INK

Format: INKEY(⟨Tastencode⟩)

Zweck:

Abfrage von gedrückten Tastenkombinationen.

Anwendung:

INKEY ermittelt, ob die Taste mit dem ⟨Tastencode⟩ allein, in Verbindung mit SHIFT, zusammen mit CTRL oder mit beiden Tasten gleichzeitig gedrückt ist.

Die folgende Tabelle schlüsselt die durch die INKEY-Anweisung ermittelten Werte nach den gedrückten Tastenkombinationen auf.

Tabelle der Tastenkombinationen

WERT	SHIFT	CTRL	TASTE
-1	–	–	–
0	–	–	*
32	*	–	*
128	–	*	*
160	*	*	*

Hierbei ist: * → Taste ist gedrückt
– → Taste ist nicht gedrückt

Beispiel:

```
10 MODE 2
20 PRINT"Bitte ENTER abwechselnd in Verbindung mit der
CTRL- bzw. SHIFT-Taste druecken"
30 IF INKEY(18)=-1 THEN 30
40 IF INKEY(18)=0 THEN T$="ENTER":GOTO 80
50 IF INKEY(18)=32 THEN T$="SHIFT + ENTER":GOTO 80
60 IF INKEY(18)=128 THEN T$="CTRL + ENTER":GOTO 80
70 T$="SHIFT + CTRL + ENTER"
80 PRINT CHR$(10)"Sie haben "T$" gedrueckt."
90 FOR I=800 TO 0 STEP -1:NEXT
100 GOTO 30
```

Vergleichen Sie hierzu: INPUT, INKEY$, INP

Format: INKEY$

Zweck:

Ermittelt eine gedrückte Taste bzw. enthält das entsprechende Zeichen.

Anwendung:

INKEY$ fragt laufend den Tastaturstatus ab, d.h., es wird geprüft, ob eine Taste gedrückt wurde oder nicht. INKEY$ liefert immer dann einen String mit der Länge Null, wenn keine Taste gedrückt wurde. Wurde eine Taste gedrückt, so kann das über die Taste eingegebene Zeichen einer Stringvariablen übergeben werden. Beachten Sie hierbei, daß das eingegebene Zeichen nicht mit der INKEY$-Anweisung auf dem Monitor ausgegeben wird.

Beispiel:

```
10 MODE 1
20 ZEICHEN$=""
30 PRINT "Programmabbruch mit ESC-Taste"
40 ZEICHEN$=INKEY$
50 IF ZEICHEN$="" THEN 40
60 PRINT ZEICHEN$;
70 GOTO 40
```

Ergebnis:

Entsprechend der gedrückten Taste wird mit der PRINT-Anweisung in Zeile 60 das dazugehörige Zeichen auf dem Monitor ausgegeben.

Vergleichen Sie hierzu auch: INPUT, INKEY

INP

Format: `INP(⟨Port⟩)`

Zweck:

Liest ein Byte vom angegebenen ⟨Port⟩ ein.

Anwendung:

`INP(⟨Port⟩)` liest ein Byte vom angewählten ⟨Port⟩ ein. ⟨Port⟩ darf Werte zwischen -32768 und 32767 annehmen. Liegt ⟨Port⟩ außerhalb dieses Bereiches, so erscheint die Fehlermeldung `Overflow`.

Der Befehl `INP` wird zur Abfrage von Ein- und Ausgabeschnittstellen genutzt.

Vergleichen Sie hierzu auch: `OUT`, `WAIT`

INPUT

Format: `INPUT [#`⟨Datenstrom⟩`,][`⟨"Abfrage"`]`⟨Variablenliste⟩

bzw.

`INPUT [#`⟨Datenstrom⟩`,][`⟨"Abfrage"⟩`,]`⟨Variablenliste⟩

Zweck:

Daten in das laufende Programm im Dialog eingeben und an Variablen übergeben.

Anwendung:

Mit der `INPUT`-Anweisung können alle möglichen Daten abgefragt und von der Tastatur eingelesen werden.

Die ⟨"Abfrage"⟩ ist nicht zwingend anzugeben, erleichtert aber den Umgang mit dem Programm.

In der ⟨Variablenliste⟩ stehen alle die Variablen, denen im laufenden Programm Werte übergeben werden sollen. Die Variablen sind in der Variablenliste mit Kommata getrennt anzugeben.

Ebenso sind die einzelnen Werte, wenn sie direkt hintereinanderfolgend abgefragt werden, bei der Eingabe im laufenden Programm mit Kommata zu trennen, da sonst die Fehlermeldung `?Redo from start` erscheint.

Wird nach ⟨"Abfrage"⟩ ein Semikolon ";" gesetzt, so erscheint auf dem Monitor nach der Abfrage ein Fragezeichen auf dem Monitor. Steht anstelle des Semikolons ein Komma ",", so wird die Ausgabe des Fragezeichens unterdrückt.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ein-/Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	.... .
#7	Bildschirmfenster 7
#9	Datacorder

Werden für ⟨Datenstrom⟩ Werte größer 9 eingegeben, so erscheint die Fehlermeldung "`Improper argument in` ⟨Zeilennummer⟩".

Beispiel 1:

```
10 INPUT "Geben Sie zwei Zahlen ein ",ZAHL1,ZAHL2
20 PRINT ZAHL1,ZAHL2
```

Bei dieser `INPUT-Anweisung wird kein Fragezeichen ausgegeben.`

Das nachstehende Beispiel zeigt:

1. Die Wirkung der Angabe "#Datenstrom" – hier die Abfrage mit der `INPUT`-Anweisung in einem bestimmten Bildschirmfenster.

2. Die Ausgabe eines Fragezeichens nach der ⟨Abfrage⟩.

3. Die Abfrage eines Strings und eines numerischen Wertes mit einer `INPUT`-Anweisung und entsprechend formulierter ⟨Variablenliste⟩.

Beispiel 2:

```
10 MODE 1
20 PAPER 3
30 CLS
40 WINDOW #1,7,33,4,18
50 PAPER #1,2
60 CLS #1
70 INK 1,0
80 LOCATE #1,1,3
90 PRINT #1,"Nachname und Personalnummer
eingeben"
100 LOCATE #1,1,6
110 INPUT #1,"Nachname";NAME$
120 INPUT #1,"Personalnummer";ZAHL1
130 LOCATE #1,1,10
140 PRINT #1,"Nachname: ";NAME$
150 LOCATE #1,1,12
160 PRINT #1,"Personalnummer: ";ZAHL1
170 LOCATE 5,2
180 PRINT "Programm mit ESC-Taste abbrechen"
190 GOTO 190
```

Vergleichen Sie hierzu auch: LINE INPUT, INKEY$, READ

INSTR

Format: `INSTR([Anfangsposition],<String>,<Suchstring>)`

Zweck:

Suchen von einem Zeichen oder einer Zeichenkette in einer vorgegebenen Zeichenkette.

Anwendung:

Mit der `INSTR([X],A$,B$)`-Anweisung können bestimmte Zeichen oder Zei chenfolgen in der Zeichenkette A$ gesucht werden. Die Startposition der gefundenen Zeichen wird dann von dem Befehl `INSTR` berechnet. Wird das Zeichen B$ nicht in A$ gefunden, so erhält man den Wert Null "0". X bestimmt die Anfangsposition des Suchens in der Zeichenkette A$.

Beispiel:

```
10 MODE 2
20 A$="Es faengt damit an, dass am Ende der Punkt fehlt"
30 I=INSTR(A$,"am")
40 J=INSTR(19,A$,"am")
50 K=INSTR(A$,"Punkt")
60 L=INSTR(A$,".")
70 PRINT:PRINT "Vorgegebene Zeichenkette:"
80 PRINT A$
90 PRINT:PRINT "gefunden:
100 PRINT " am"," am"," Punkt"," ."
110 PRINT:PRINT "In Position:":PRINT I,J,K,L
120 PRINT:PRINT:END
```

Ergebnis:

```
Vorgegebene Zeichenkette:
Es faengt damit an, dass am Ende der Punkt fehlt

gefunden:
am      am      Punkt   .

in Position:
12      26      38      0
```

Vergleichen Sie hierzu auch: MID$, LEFT$, RIGHT$, SPACE$

INT

Format: INT(X)

Zweck:

Ergibt die größte ganze Zahl, die kleiner oder gleich X ist.

Anwendung:

INT(X) liefert als Ergebnis die größte ganze Zahl, die kleiner oder gleich X ist. Die INT-Anweisung ist ähnlich der FIX-Anweisung, nur daß hierbei für negative Werte um 1 kleinere Ergebnisse anfallen, falls diese nicht vorher schon ganzzahlig sind.

Beispiel:

```
PRINT INT(9.999)            ergibt 9
PRINT INT(.765)             ergibt 0
PRINT INT(-55.2)          ergibt -56
```

Vergleichen Sie hierzu auch: CINT, FIX, ROUND

JOY

Format: JOY (⟨Integerzahl⟩)

Zweck:

Ermittelt den Wert des angegebenen Joysticks.

Anwendung:

Die JOY-Anweisung liefert die Stellung des Joysticks, wobei hier bestimmte Bits für definierte Stellungen gesetzt sind. Die ⟨Integerzahl⟩ gibt den Joystick an und kann den Wert "0" für den ersten und "1" für den zweiten Joystick annehmen.

Joysticktabelle

Bit	Dezimalwert	Joystickstellung
0	1	hoch
1	2	runter
2	4	links
3	8	rechts
4	16	F 2
5	32	F 1

Als zweiter Joystick kann auch die Tastatur des ⟩⟩ CPC 464 ⟨⟨ benutzt werden. Nachstehend hierfür die Tastenangabe:

Bit	Dezimalwert	Joystickstellung	Tastencode	Zeichen
0	1	hoch	48	6
1	2	runter	49	5
2	4	links	50	R
3	8	rechts	51	T
4	16	F 2	52	G
5	32	F 1	53	F

Vergleichen Sie hierzu auch: INKEY

Format:

KEY ⟨Integerausdruck⟩, [CHR$(N)+]⟨Stringausdruck⟩[CHR$(N)]

Zweck:

Belegt die Funktionstasten mit einem Text oder einer Kommandofolge und ordnet diese der entsprechenden Taste zu.

Anwendung:

Die KEY-Anweisung ermöglicht eine benutzerdefinierte Funktionstastenbelegung. Die Tasten mit dem Tastencode 128 bis 159 stehen für eigene Belegungen zur Verfügung. Die Anzahl der Zeichen darf insgesamt 120 nicht überschreiten.

Beispiel:

```
KEY 138,"MODE 1:LIST"+CHR$(13)
```

Ergebnis:

Wird die Taste mit dem Code 138 gleich Punkt im Ziffernblock gedrückt, so wird das Bildschirmfenster gelöscht und ein im Arbeitsspeicher befindliches Programm gelistet.

Vergleichen Sie hierzu auch: KEY DEF

KEY DEF

Format:

KEY DEF ⟨Tastencode⟩,⟨repeat⟩,⟨ASCII-Wert⟩
[,⟨ASCII-Wert⟩][,⟨ASCII-Wert⟩]

Zweck:

Verändert den ASCII-Wert einer Taste auf einen anderen vom Benutzer gewünschten ASCII-Wert.

Anwendung:

Die KEY DEF-Anweisung definiert einzelne Tasten in das vom Benutzer gewünschte Zeichen um. Die neu zu belegende Taste wird im Tastencode, das gewünschte Zeichen wird mit seinem ASCII-Wert angegeben. Wird für ⟨repeat⟩ eine Null angegeben, so ist die Wiederholfunktion der Taste ausgeschaltet. Ist sie erwünscht, so ist anstelle der Null eine Eins zu setzen. Es sind insgesamt drei verschiedene ASCII-Wert Angaben möglich. Der erste ASCII-Wert wird ausgegeben, wenn die Taste allein gedrückt wird, der zweite, wenn sie in Kombination mit der SHIFT-Taste und der dritte angegebene ASCII-Wert, wenn die Tastenkombination CTRL plus der Taste mit dem vorgegebenen ⟨Tastencode⟩ gleichzeitig gedrückt werden.

Beispiel:

```
10 KEY DEF 66,0,127
20 I=0
30 I=I+1
40 PRINT I,
50 GOTO 30
```

Ergebnis:

Sie haben hiermit ein Programm, das sich nicht mehr durch Drücken der ESC-Taste abbrechen läßt, da auf der ESC-Taste nun der ASCII-Wert 127 liegt.

Vergleichen Sie hierzu auch: KEY

LEFT$

Format: LEFT$(⟨String⟩,⟨Anzahl der Zeichen⟩)

Zweck:

Überträgt den linken Teil einer Zeichenkette.

Anwendung:

LEFT$(A$,I) überträgt von links I Zeichen aus der Zeichenkette A$ in die Zielvariable. Der Wert von I muß zwischen 0 und 255 liegen. Ist I größer als die Anzahl der Zeichen der Zeichenkette A$, so wird diese vollständig übertragen. Ist I=0 so wird kein Zeichen von A$ übertragen. Die Fehlermeldung Improper argument ⟨Zeilennummer⟩ erscheint, wenn I negativ oder größer 255 ist.

Beispiel:

```
10 MODE 2
20 BORDER 20
30 C$=STRING$(6,42)
40 A$=" CPC 464 COLOUR COMPUTER mit dem ausgereiften
BASIC"
50 B$=LEFT$(A$,25)
60 LOCATE 11,8:PRINT C$;B$;C$
70 D$=LEFT$(A$,0)
80 LOCATE 11,10:PRINT C$;D$;C$
90 E$=LEFT$(A$,80)
100 LOCATE 10,12:PRINT E$
110 END
```

Ergebnis:

```
     ***** CPC 464 COLOUR COMPUTER *****

     **********

     CPC 464 COLOUR COMPUTER mit dem ausgereiften BASIC
```

Vergleichen Sie hierzu auch: RIGHT$, MID$, STRING$, SPACE$

LEN

Format: `LEN(⟨String⟩)`

Zweck:

Ermittelt die Anzahl der Zeichen einer Zeichenkette.

Anwendung:

`LEN(A$)` ermittelt die Anzahl der Zeichen in der Zeichenkette A$ ein schließlich aller blanks (Leerzeichen) und nicht abdruckbarer Zeichen.

Beispiel:

```
10 A$="Eine Schlittenfahrt im Winter"
20 L=LEN(A$)
30 PRINT L
```

Ergebnis:

```
29
```

Vergleichen Sie hierzu auch: `RIGHT$, LEFT$, MID$, VAL`

Format: `LET`

Zweck:

Ordnet den Wert eines Ausdrucks einer Variablen zu.

Anwendung:

Das Wort `LET` dient dazu, im Rahmen einer Anweisung einer Variablen den Wert eines Ausdrucks zuzuweisen. `LET` wird jedoch hier nur noch als Option benutzt, um die Kompatibilität zu älteren BASIC-Dialekten zu gewährleisten. Man kann `LET` ohne negative Wirkung weglassen. Der Variablen links vom Gleichheitszeichen wird der Wert des Ausdrucks rechts vom Gleichheitszeichen zugewiesen.

Beispiele:

```
LET A=B*C
```

kann unter diesem BASIC-Interpreter so programmiert werden:

```
A=B*C
```

und

```
LET A$="CPC 464 - 64k Colour COMPUTER"
```

kann unter diesem hochwertigen BASIC-Interpreter so programmiert werden:

```
A$="CPC 464 - 64k Colour COMPUTER"
```

LINE INPUT

Format:

LINE INPUT [#⟨Datenstrom⟩,][;][⟨Abfrage⟩;]⟨Variable$⟩

Zweck:

Daten zeilenweise in das laufende Programm im Dialog eingeben und an Variable übergeben.

Anwendung:

Mit der LINE INPUT-Anweisung werden alle eingegebenen Zeichen bis zum nächsten RETURN der ⟨Variable$⟩ übergeben. Die Eingabe darf maximal 255 Zeichen umfassen und muß durch Drücken der RETURN- bzw. der ENTER- Taste abgeschlossen werden. Wird das Semikolon ";" direkt nach der INPUT-Anweisung gesetzt, so wird der automatische Zeilenvorschub von der LINE-INPUT-Anweisung unterdrückt.

Das zwischen ⟨Abfrage⟩ und ⟨Variable⟩ befindliche Semikolon ";" kann durch ein Komma "," ersetzt werden. Es hat hier nicht die Bedeutung, wie Sie sie von der INPUT-Anweisung kennen. Das Fragezeichen wird in keinem Fall ausgegeben, d.h., es muß an die ⟨Abfrage⟩ mit angehängt werden, falls es gewünscht wird.

Der Vorteil dieser Anweisung gegenüber der INPUT-Anweisung ist der, daß unkundige Programmbenutzer mehrere Eingaben nacheinander eingeben können, ohne hierbei auf die Trennung achten zu müssen. Zudem müssen bei vorangestellten Leerzeichen bzw. enthaltenen Kommata keine Anführungszeichen gesetzt werden.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ein-/Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	....
#7	Bildschirmfenster 7
#9	Datacorder

Beispiel:

```
10 MODE 1
20 LINE INPUT #0,"Geben Sie einen Satz ein: ";SATZ$
30 PRINT #0,SATZ$
```

Wird Zeile 20 wie folgt ersetzt:

```
20 LINE INPUT #0,;"Geben Sie einen Satz ein: ";SATZ$
```

so wird der automatische Zeilenvorschub unterdrückt.

Vergleichen Sie hierzu auch: INPUT, INKEY$, READ

LIST

Format:

LIST

bzw.

LIST [⟨Zeilennummer⟩] [-] [⟨Zeilennummer⟩] [,#⟨Datenstrom⟩]

Zweck:

Listen vom im Arbeitsspeicher befindlichen Programm bzw. von Programmteilen.

Anwendung:

LIST listet das gesamte im Arbeitsspeicher des ⟩⟩ CPC 464 ⟨⟨ vorhandene Programm auf dem Monitor aus. Die zweite Form des LIST-Kommandos ermöglicht ein gezieltes Listen von Zeilenbereichen. Wird ⟨Datenstrom⟩=8 angegeben, so erfolgt der Ausdruck auf dem angeschlossenen Drucker.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ein-/Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	... .
#8	Drucker
#9	Datacorder

Vor der Ausgabe mit der LIST- #9-Anweisung auf dem Datacorder muß eine Datei mit der OPENOUT-Anweisung eröffnet werden.

Beispiele:

```
LIST 195
```
listet die Zeile 195 auf.

```
LIST 188-
```
listet alle Programmzeilen ab Zeile 188 einschließlich auf.

```
LIST -80
```
listet alle Programmzeilen bis zur Zeile 80 einschließlich auf.

```
LIST 105-500
```
listet alle Programmzeilen von 105 bis 500 einschließlich auf.

Das Auflisten des Programms kann durch ein einmaliges Drücken der ESC-Taste `*Break*` unterbrochen werden und mit dem Drücken der LEER-Taste wieder weitergeführt werden. Ein zweimaliges Drücken der ESC-Taste `*Break*` führt in den Kommandomodus von BASIC zurück.

LOAD

Format:

LOAD ""

bzw.

LOAD "⟨Programmname⟩"

Zweck:

Laden von Programmen bzw. Dateien vom Datacorder.

Anwendung:

LOAD lädt Programme bzw. Dateien von Kassette in den Arbeitsspeicher ein. Der ⟨Programmname⟩ darf maximal eine Länge von 16 Buchstaben aufweisen, wobei hier nicht zwischen Groß- und Kleinschreibung unterschieden wird.

Beispiel 1:

LOAD ""

lädt das nächstliegende Programm von Kassette in den Arbeitsspeicher ein.

Beispiel 2:

Zum Laden von Binärdateien kann zusätzlich eine Anfangsadresse im Speicher angegeben werden, ab der die Binärdatei abgelegt wird. Das Schema hierfür ist:

LOAD "Dateiname",Anfangsadresse

Beispiel 3:

```
LOAD "!Abrechnung"
```

es wird das Programm "Abrechnung" in den Arbeitsspeicher des Schneider-Computers ⟩⟩ CPC464 ⟨⟨ geladen, ohne hierbei Bedienungshinweise auf dem Monitor auszugeben.

Vergleichen Sie hierzu auch:

SAVE, CAT, RUN "⟨Programmname⟩", MERGE, CHAIN MERGE, CHAIN

LOCATE

Format: LOCATE [#⟨Datenstrom⟩,]⟨SpalteX⟩,⟨ZeileY⟩

Zweck:

Positionieren des Textcursors.

Anwendung:

LOCATE dient der Cursorsteuerung und setzt den Textcursor an eine neue Bildschirmfensterposition. ⟨SpalteX⟩ bezeichnet die Spalte und ⟨ZeileY⟩ die Zeile. ⟨SpalteX⟩ und ⟨ZeileY⟩ sollten ganzzahlige, numerische Ausdrücke sein, die in Abhängigkeit von der Größe des Bildschirmfensters zu wählen sind.

Die linke obere Ecke des Monitorfensters besitzt hierbei die Koordinaten ⟨SpalteX⟩=1 und ⟨ZeileY⟩=1. Sollten Sie die ⟨SpalteX⟩ und ⟨ZeileY⟩ Werte variabel übergeben, so ist darauf zu achten, daß keine negativen Werte entstehen.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	....
#7	Bildschirmfenster 7

Werte größer 7 ergeben die Fehlermeldung Improper argument.

Ein Beispiel für die Anwendung der Anweisung LOCATE #⟨Datenstrom⟩ finden Sie bei der PRINT-Anweisung.

Beispiel:

```
10 Mode 2
20 X=10-5:Y=9-3
30 LOCATE X,Y:PRINT "Dieser Text beginnt in
Spalte";X;"und steht in der Zeile";Y
40 END
```

Vergleichen Sie hierzu auch: WINDOW, MOVE, WINDOW SWAP

LOG

Format: `LOG(⟨numerischer Ausdruck⟩)`

Zweck:

Berechnen des natürlichen Logarithmus.

Anwendung:

`LOG(`⟨numerischer Ausdruck⟩`)` ermittelt den natürlichen Logarithmus des ⟨numerischen Ausdrucks⟩. Da die Logarithmusfunktion nur für positive Werte definiert ist, erhält man bei negativen Werten die Fehlermeldung `Overflow` mit einem Ausdruck des negativen Wertes auf dem Monitor.

Beispiel:

```
? LOG(1.7E+38)
```

Ergebnis:

```
88.0288618
```

Vergleichen Sie hierzu auch: `EXP, LOG10`

LOG10

Format: `LOG10(`⟨numerischer Ausdruck⟩)

Zweck:

Berechnen des Logarithmus zur Basis 10.

Anwendung:

`LOG10(`⟨numerischer Ausdruck⟩`)` berechnet den dekadischen Logarithmus (Basis 10) des ⟨numerischen Ausdrucks⟩ und ergibt eine reelle Zahl.

Beispiel:

```
? LOG10(100)
```

Ergebnis:

```
2
```

Vergleichen Sie hierzu auch: `LOG, EXP`

LOWER$

Format: `LOWER$(⟨String⟩)`

Zweck:

Wandelt Großbuchstaben in Kleinbuchstaben um.

Anwendung:

Die `LOWER$(A$)`-Anweisung wandelt die Großbuchstaben der Zeichenkette A$ in Kleinbuchstaben um. Dies ist nützlich bei der Auswertung von eingegebenen Zeichen nach einer `INPUT`-Anweisung.

Beispiel:

```
PRINT LOWER$("DieSEs isT eINE vErsuCHSZEICHenKEtTe!")
```

Ergebnis:

```
dieses ist eine versuchszeichenkette!
```

Vergleichen Sie hierzu auch: `UPPER$`

Format: `MAX` (⟨Liste von numerischen Variablen⟩)

Zweck:

Ermitteln des größten Wertes einer Liste.

Anwendung:

`MAX` ermittelt aus der ⟨Liste der numerischen Variablen⟩ bzw. Konstanten das Maximum, d.h., den größten Wert. Dieser Wert kann direkt ausgegeben oder einer Variablen übergeben werden.

Beispiel:

```
10 N=100
20 PRINT MAX (8,99,15.45783,48,N)
```

Ergebnis:

```
100
```

Vergleichen Sie hierzu auch: `MIN`

MEMORY

Format: MEMORY ⟨Adresse⟩

Zweck:

Setzen der höchsten für BASIC benutzbaren Speicheradresse.

Anwendung:

Mit MEMORY kann die Größe des für BASIC frei verfügbaren Speicherplatzes gesetzt werden, indem man die höchste Adresse angibt, die BASIC benutzen darf. Diese Adresse kann mit HIMEM abgefragt werden. Der sich daraus ergebende freie Speicherplatz für BASIC kann mit FRE abgefragt werden.

Vergleichen Sie hierzu auch: HIMEM, FRE

MERGE

Format: `MERGE ["`⟨Programmname⟩`"]`

Zweck:

Einmischen der Programme vom Datacorder in ein im Arbeitsspeicher befindliches Programm.

Anwendung:

Die `MERGE`-Anweisung mischt ein auf Compact-Kassette gespeichertes Teilprogramm mit dem angegebenen ⟨Programmnamen⟩ in ein im Arbeitsspeicher des ⟩⟩ CPC 464 ⟨⟨ vorhandenes Teilprogramm ein. Entfällt die Angabe ⟨Programmname⟩, so wird das erste auf der Kassette gefundene Programm eingemischt.

Ist das erste Zeichen von ⟨Programmname⟩ ein Ausrufezeichen "!", so werden die vorher auf Kassette gefundenen Programmnamen nicht auf dem Monitor ausgegeben und alle Bedienungshinweise für den Kassettenleseprozeß unterdrückt.

Liegen bei beiden Teilprogrammen teilweise gleiche Zeilennummern vor, so werden die identischen Zeilennummern und deren Anweisungen von dem eingemischten Teilprogramm überschrieben. Beachten Sie dieses, und benutzen Sie vorher die `RENUM`-Anweisung, um das im Arbeitsspeicher befindliche Teilprogramm entsprechend zu numerieren.

Mit `SAVE "Programmname", P` abgespeicherte Programme lassen sich nicht mit der `MERGE`-Anweisung mischen.

Beispiel:

```
10 '****************************************
20 '* *
30 '* Benchmark - Test A *
40 '* *
50 '****************************************
60 REM
```

Geben Sie diese Kommentarzeilen ein und speichern Sie diese dann unter dem Programmnamen "TEIL1" auf der Kassette ab. Die Kommentarzeilen sind nun unter dem Namen "TEIL1" gespeichert.

Geben Sie im Kommandomodus NEW ein und anschließend die LIST-Anweisung. Die eingegebenen Zeilen sind nun im Arbeitsspeicher gelöscht. Geprüft wurde mit der LIST-Anweisung.

Tippen Sie nun das folgende kleine Programm ein:

```
10 REM BENCHMARK - TEST A
20 PRINT "START"
30 FOR J=1 TO 1000
40 NEXT J
50 PRINT "ENDE"
60 END
```

Im Kommandomodus nun bitte **RENUM 60.** Prüfen Sie nun mit der **LIST**-Anweisung die Zeilennumerierung. Sie lautet:

    60,70,...,110.

Geben Sie folgende Anweisung im Kommandomodus ein, nachdem das Kassettenband zurückgespult ist:

```
MERGE "TEIL1"
```

Ist der Einlesevorgang beendet, so lassen Sie sich den Inhalt des Arbeitsspeichers mit der LIST-Anweisung ausgeben.

```
LIST
```

Ergebnis:

```
10 '***************************************
20 '* *
30 '* Benchmark - Test A *
40 '* *
50 '***************************************
60 REM
70 PRINT "START"22 80 FOR J=1 TO 1000
90 NEXT J
100 PRINT "ENDE"
110 END
```

Sie haben beide Teilprogramme zu einem Programm zusammengemischt.

Beachten Sie hierbei die Zeile 60, sie wurde durch Programm "TEIL1" überschrieben.

Vergleichen Sie hierzu auch:

LOAD, RUN ("Programmname"), CHAIN, CHAIN MERGE, SAVE

MID$

Format: MID$(⟨String⟩,⟨Anfangsposition⟩[,⟨Anzahl der Zeichen⟩])

Zweck:

Übertragen von Zeichenkettenteilen aus einer vorgegebenen Zeichenkette.

Anwendung:

MID$ entnimmt oder ersetzt aus dem ⟨String⟩ ein Zeichen oder eine Zeichenkette.

Vier Anwendungsfälle lassen sich bei dieser Stringfunktion unterscheiden:

1. MID$(A$,I)

   überträgt die Zeichenkette A$, beginnend mit dem I-ten Zeichen.

2. MID$(A$,I,J)

   überträgt J Zeichen der Zeichenkette A$ und beginnt dabei mit dem I-ten Zeichen.

3. MID$(A$,I)=Y$

   Die Zeichenkette A$ wird vom I-ten Zeichen an durch die Zeichenkette Y$ ersetzt. Alle Zeichen von Y$ werden übertragen, sofern A$ lang genug ist.

4. MID$(A$,I,J)=Y$

   Mit dem I-ten Zeichen beginnend, werden J Zeichen von A$ durch Zeichen von Y$ ersetzt.

Vergleichen Sie hierzu auch: LEFT$, RIGHT$, STRING$

Format: MIN (⟨Liste von numerischen Variablen⟩)

Zweck:

Ermitteln des kleinsten Wertes einer Liste.

Anwendung:

MIN ermittelt aus einer Anzahl numerischer Variablen oder Konstanten das Minimum, d.h., den kleinsten Wert.

Beispiel:

```
10 MODE 1
20 LOCATE 5,5
30 PRINT "Geben Sie bitte zwei Zahlen ein"
40 LOCATE 5,7: INPUT "Die erste Zahl lautet :";Z1
50 LOCATE 5,8: INPUT "Die zweite Zahl lautet:";Z2
60 X=MIN(Z1,Z2,40)
70 IF X=40 GOTO 90
80 LOCATE 5,10:PRINT "Die kleinste Zahl ist";X:END
90 LOCATE 5,10
100 PRINT "Die in Zeile 60 vorgegebene Zahl"
110 LOCATE 4,12:PRINT X;"ist kleiner !"
```

Ergebnis:

```
Geben Sie bitte zwei Zahlen ein

Die erste Zahl lautet:? 34
Die zweite Zahl lautet:? 67

Die kleinste Zahl ist 34
```

Vergleichen Sie hierzu auch: `MAX`

Zweck: Berechnung des Modulus

Anwendung:

MOD berechnet den Rest, der bei einer Division ensteht, wenn nur ganze Zahlen als Ergebnis angenommen werden. Das Ergebnis der Berechnung mit MOD ist immer eine Intergerzahl.

Beispiel:

```
?5 MOD 2

1
```

Beispiel:

```
10 FOR I=1 TO 100
20 PRINT I
30 IF I MOD 10 = 0 THEN LINE INPUT A$
40 PRINT A$
50 NEXT I
```

Vergleichen Sie hierzu auch: Mathematische Operatoren

MODE

Format: MODE X

Zweck:

Änderung des Bildschirmformates auf 20, 40 oder 80 Zeichen pro Zeile.

Anwendung:

Mit der MODE-Anweisung können drei unterschiedliche Bildschirmformate aufgerufen werden, d.h., die Spaltenanzahl kann vorgegeben werden. Die Zahl X kann hierbei die Werte von null "0" bis zwei "2" annehmen.

Hierbei ist:

MODE 0 Das Bildschirmformat zu 20 Spalten.
MODE 1 Das Bildschirmformat zu 40 Spalten.
MODE 2 Das Bildschirmformat zu 80 Spalten.

Nach dem Einschalten des ⟩⟩ CPC 464 ⟨⟨ befindet sich dieser automatisch in MODE 1. Die Hintergrundfarbe und der Bildschirmrand sind blau und die Zeichen hellgelb. Die MODE-Anweisung löscht den mit CHR$(22)CHR$(1) eingeschalteten "Transparent-Modus".

Die auf dem Monitor maximal nebeneinander darstellbaren Farben werden durch die MODE-Anweisung beeinflußt.

Hierbei sind:

In `MODE 0` maximal 16 der 27 möglichen Farben,
in `MODE 1` maximal 4 der 27 möglichen Farben,
in `MODE 2` maximal 2 der 27 möglichen Farben

nebeneinander verfügbar.

Vergleichen Sie hierzu auch: `PAPER, INK, PEN`

MOVE

Format: MOVE ⟨X-Koordinate⟩,⟨Y- Koordinate⟩

Zweck:

Positionierung des Graphik-Cursors auf eine Koordinate.

Anwendung:

MOVE setzt den Graphikcursor auf die nach der MOVE-Anweisung angegebenen Koordinaten. Durch die CLG-Anweisung werden diese Koordinaten wieder auf die Ursprungskoordinate 0,0 des Graphikbildschirmfensters gesetzt. Das CLS-Anweisung verändert die Graphikcursorposition nicht. Mit XPOS bzw. YPOS kann die aktuelle Graphikcursorposition abgefragt werden.

Beispiel:

```
10 MODE 1
20 BORDER 18
30 MOVE 214,159
40 CLS
50 DRAW 380,250
60 END
```

Kommentar:

In Zeile 30 wird der Graphikcursor auf die Position 214,159 gesetzt. Nach dem CLS-Befehl bleibt diese Position erhalten. Bitte ersetzen Sie jetzt Zeile 40 durch:

```
40 CLG
```

Hierdurch wird die Graphikcursorposition, die durch MOVE gesetzt wurde, gelöscht.

Vergleichen Sie hierzu auch:

MOVER, ORIGIN, PLOT, PLOTR, DRAW, DRAWR, TEST, TESTR, XPOS, YPOS

MOVER

Format: `MOVER` ⟨rel. X-Koordinate⟩,⟨rel. Y-Koordinate⟩

Zweck:

Relative Positionierung des Graphik-Cursors.

Anwendung:

`MOVER` setzt den Graphikcursor relativ zur aktuellen Graphikcursorposition. Die X-Koordinate der neuen Position des Graphikcursors wird bestimmt durch die X-Koordinate der aktuellen Graphik-Cursorposition + ⟨rel. X-Koordinate⟩. Die Y-Koordinate berechnet sich entsprechend der X-Koordinate. Durch die `CLG`-Anweisung werden diese Koordinaten wieder auf die Ursprungskoordinate 0,0 des Graphikbildschirmfensters gesetzt. Die `CLS`-Anweisung verändert die Graphikcursorposition nicht.

Beispiel:

```
10 MODE 1
20 BORDER 18
30 MOVE 0,200
40 DRAWR 150,0
50 FOR I=0 TO 1000:NEXT I
60 MOVER 0,-50
70 DRAWR 150,0
80 LOCATE 1,14:PRINT "MOVE"
90 LOCATE 10,17:PRINT"MOVER"
100 LOCATE 1,24:PRINT"Programm bitte mit ESC-Taste
abbrechen."
110 GOTO 110
```

Kommentar:

In Zeile 60 wird der Graphikcursor um den Wert 50 nach unten gesetzt. Hierdurch erklärt sich der Startpunkt der DRAWR-Anweisung in Zeile 70.

Vergleichen Sie hierzu auch:

MOVER, ORIGIN, PLOT, PLOTR, DRAW, DRAWR, TEST, TESTR, XPOS, YPOS

NEW

Format: NEW

Zweck:

Löschen des im Arbeitsspeichers befindlichen Programms und der benutzten Variablen.

Anwendung:

NEW wird im Kommando-Modus eingegeben, bevor ein neues BASIC-Programm in den Arbeitsspeicher geschrieben wird. Es löscht das im Speicher vorhandene Programm und alle Variablen. Die Funktionstastenbelegung und das Monitorbild bleiben bei diesem Befehl erhalten. Der Inhalt des Monitorbildes kann aber lediglich mit der "COPY"-Editierfunktion gerettet werden. Die NEW-Anweisung ist immer vor der Eingabe eines neuen BASIC-Programms einzugeben. Der BASIC-Interpreter meldet sich nach der Eingabe dieser Anweisung im Kommandomodus.

Verwenden Sie diesen Befehl mit der entsprechenden Vorsicht, da er in seiner Wirkung einem Ausschalten des Computers entspricht.

Vergleichen Sie hierzu auch: CLS, CLG, CLEAR, DELETE

Hinweis:

NEXT darf in einem Programm niemals allein stehen. Zu einem NEXT gehört immer ein FOR. Deshalb ist NEXT unter dem Stichwort FOR behandelt.

Vergleichen Sie hierzu: FOR ... NEXT

NOT

Logische Verknüpfung: NOT

Zweck:

Ableiten von Entscheidungen für den Programmablauf.

Anwendung:

Die logische Verknüpfung NOT a ist immer dann genau wahr, wenn der Ausdruck a den Wert 0 besitzt, d.h., falsch ist.

Wahrheitstafel: WAHR = 1; FALSCH = 0

a	NOT a
1	0
0	1

Vergleichen Sie hierzu auch: AND, OR

ON BREAK GOSUB

Format: `ON BREAK GOSUB ⟨Zeilennummer⟩`

Zweck:

Ausführen eines Unterprogrammsprungs nach Drücken der ESC-Taste.

Anwendung:

Mit Hilfe der Anweisung `ON BREAK GOSUB` kann in einem BASIC-Programm verhindert werden, daß das Programm durch zweimaliges Drücken der ESC-Taste abgebrochen wird. Bei Verwendung von `ON BREAK GOSUB` wird beim zweiten Drücken der ESC-Taste statt der üblichen Meldung `"*Break*"` ein Sprung in ein vom Programmierer vorgegebenes Unterprogramm ausgeführt. Dieses Unterprogramm muß mit RETURN abgeschlossen werden. Ein einmaliges Drücken bewirkt nur eine Unterbrechung des Programmablaufs. Ein zweimaliges Drücken der ESC-Taste verzweigt in das durch ⟨Zeilennummer⟩ angegebene Unterprogramm.

Beispiel:

```
10 MODE 1
20 PRINT"Das Programm ist nicht abzubrechen."
30 ON BREAK GOSUB 100
40 FOR I=1 TO 100:PRINT I;:NEXT I
50 GOTO 40
100 PRINT"Sie haben 2 mal die ESC-Taste gedrueckt."
110 PRINT"Es ist aber kein Abbruch moeglich."
120 RETURN
```

Vergleichen Sie hierzu auch: `ON BREAK STOP, RETURN`

ON BREAK STOP

Format: ON BREAK STOP

Zweck:

Inaktivieren der Anweisung ON BREAK GOSUB.

Anwendung:

ON BREAK STOP wird verwendet, um die Anweisung ON BREAK GOSUB in einem Programm zu inaktivieren. Dies kann angewendet werden, um nur in bestimmten Programmteilen oder nach einer bestimmten Anweisungsfolge einen Abbruch zu gestatten.

Beispiel:

```
10 MODE 2
15 ON BREAK GOSUB 100
20 PRINT"Das Programm kann nur mit der Tastenfolge:"
25 PRINT:PRINT TAB(10)"2 mal ESC plus E plus 2 mal ESC-
Taste"
26 PRINT:PRINT"abgebrochen werden."
30 FOR I=32 TO 100
40 PRINT CHR$(I);
50 NEXT I
60 GOTO 30
90 REM **** Unterprogramm fuer Abbruch *** 100 A$=IN-
KEY$:IF A$="" THEN 100
105 IF A$="E" OR A$="e" THEN ON BREAK STOP
110 RETURN
```

Vergleichen Sie hierzu auch: ON BREAK GOSUB

ON ERROR GOTO

Format: `ON ERROR GOTO` ⟨Zeilennummer⟩

Zweck:

Erkennen eines Fehlers und fehlerbedingter Programmsprung in eine Fehlerbehandlungsroutine.

Anwendung:

`ON ERROR GOTO` wird zur Fehlerbehandlung in BASIC-Programmen genutzt. Diese können Bedienungsfehler sein, die mit der Anweisung `ON ERROR GOTO` aufgefangen werden können. Tritt ein Fehler nach dem Programmstart auf, so wird das Programm nicht abgebrochen, sondern bei der ⟨Zeilennummer⟩ fortgesetzt. Dort wird entschieden, ob das Programm noch sinnvoll fortgesetzt werden kann, oder abgebrochen werden muß. Es läßt sich mit `ERR` die Fehlerart abfragen und ein entsprechender Hinweis in deutscher Sprache ausgeben. Kann das Programm noch fortgesetzt werden, so muß am Ende der Fehlerbehandlungsroutine eine der folgenden Anweisungen stehen:

a) `RESUME`

Das Programm soll nach einer Fehlerbeseitigung in derjenigen Zeile fortgesetzt werden, in der der Fehler aufgetreten ist.

b) `RESUME NEXT`

Das Programm soll nach der Fehlerbehandlung mit der Anweisung fortgesetzt werden, die auf die Fehler-Anweisung folgt.

c) `RESUME` ⟨Zeilennummer⟩

Das Programm soll bei der angegebenen ⟨Zeilennummer⟩ fortgesetzt werden.

Beispiel:

```
10 MODE 1
20 ON ERROR GOTO 100
30 INPUT"Bitte geben Sie eine Zahl ein: ",Z
40 WERT=WERT/Z
50 PRINT"Diese Eingabe war korrekt"
60 END
100 REM *** FEHLERBEHANDLUNG ***
110 PRINT"Bitte keine Null eingeben"
120 RESUME 30
```

Vergleichen Sie hierzu auch: RESUME, ERR, ERL

Format: ON ⟨Ausdruck⟩GOSUB⟨Zeilennummernliste⟩

Zweck:

Ausführen eines berechneten Unterprogrammsprungs.

Anwendung:

Mit ON GOSUB kann ein berechneter Sprung zu bestimmten Unterprogrammen ausgeführt werden, die mit RETURN abgeschlossen sein müssen. Das Programm springt in Abhängigkeit vom numerischen Wert des ⟨Ausdrucks⟩ zu den Unterprogrammen, d.h., wenn der Wert gleich 1 ist, so wird das Unterprogramm, das in der ⟨Zeilennummernliste⟩ mit der ersten Zeilennummer angegeben ist, angesprungen.

Wird der numerische Wert z.B. gleich 5, so wird entsprechend das Unterprogramm, das in der ⟨Zeilennummernliste⟩ mit der fünften Zeilennummer angegeben ist, angesprungen.

Ist der Wert des Integerausdrucks beim Erreichen der Anweisung ON GOSUB gleich Null oder größer als die Zahl der angegebenen Zeilennummern, so wird die Anweisung übersprungen.

Beispiel:

```
10 MODE
2 20 LOCATE 1,5
30 INPUT"Bitte geben Sie eine Zahl zwischen 1 und 3
ein:",Z
40 ON Z GOSUB 100,200,300
50 END
100 PRINT:PRINT"1 wurde eingegeben ";
110 PRINT" → Sprung in das Unterprogramm ab Zeile 100."
120 RETURN
200 PRINT:PRINT"2 wurde eingegeben ";
210 PRINT" → Sprung in das Unterprogramm ab Zeile 200."
220 RETURN
300 PRINT:PRINT"3 wurde eingegeben ";
310 PRINT" → Sprung in das Unterprogramm ab Zeile 300."
320 RETURN
```

Vergleichen Sie hierzu auch: ON GOTO, GOSUB

ON GOTO

Format: `ON`⟨Ausdruck⟩ `GOTO` ⟨Zeilennummernliste⟩

Zweck:

Bewirkt einen berechneten Programmsprung in eine bestimmte Programmzeile.

Anwendung:

Mit `ON GOTO` kann ein berechneter Sprung zu bestimmten Zeilen des Programms in Abhängigkeit vom numerischen Wert des ⟨Ausdrucks⟩ durch geführt werden. Ist dieser Wert = n, so wird zur n-ten Zeilennummer der ⟨Zeilennummernliste⟩ gesprungen und dort der Programmablauf fortgesetzt. Ist der Wert des ⟨Ausdrucks⟩ beim Erreichen der Anweisung `ON GOTO` gleich null `"0"` oder größer als die Anzahl der Zeilennummern der ⟨Zeilennummernliste⟩, so wird die Anweisung übersprungen. Der Wert des ⟨Ausdrucks⟩ muß zwischen 0 und 255 liegen, da sonst die Fehlermeldung `Improper argument in` ⟨Zeilennummer⟩ ausgegeben wird.

Beispiel:

```
10 MODE 2
15 TEXT1$="Sie haben eine":TEXT2$="eingegeben."
20 INPUT "Geben Sie eine Zahl zwischen 0 und 3 ein: ";Z
25 ON Z GOTO 100,200,300
30 PRINT TEXT1$;Z;TEXT2$
40 END
100 PRINT TEXT1$;Z;TEXT2$:GOTO 40
200 PRINT TEXT1$;Z;TEXT2$
250 END
300 PRINT TEXT1$;Z;TEXT2$
310 PRINT STRING$(27,"")
320 END
```

Vergleichen Sie hierzu auch: ON GOSUB, GOTO

ON SQ GOSUB

Format: `ON SQ (`<Kanalnummer>`) GOSUB` <Zeilennummer>

Zweck:

Prüfen des Tonkanals und hiervon bedingter Unterprogrammsprung.

Anwendung:

Mit Hilfe dieser Anweisung wird ein Interrupt eingeschaltet, wenn es einen freien Platz in der Warteschlange der angegebenen <Kanalnummer> gibt. Das Programm führt dann einen Sprung zu einem Unterprogramm mit der angegebenen <Zeilennummer> aus.

Für <Kanalnummer> sind folgende Werte anzugeben:

Kanalnummer	Kanal
1	A
2	B
4	C

Vergleichen Sie hierzu auch: `SOUND`, `SQ`

OPENIN

Format: `OPENIN` "Dateiname"

Zweck:

Eröffnen eines Eingabefiles von Kassette.

Anwendung:

`OPENIN` eröffnet einen Datenfile, legt einen Datenpuffer von 2 k an und liest die abgespeicherten Daten als Datenblöcke von Kassette mit dem laufenden Programm ein. Falls der erste Buchstabe des Dateinamens ein Ausrufezeichen "!" ist, so werden keine Anweisungen zur Bedienung des Datacorders auf dem Monitor ausgegeben.

Beachten Sie bei der Anwendung des Beispielprogramms, daß Sie dieses vorher mit der `MERGE`-Anweisung mit dem Beispielprogramm bei der `OPENOUT`-Anweisung mischen können. Bevor Sie dieses Programm nutzen, sollten Sie unbedingt eine Daten-Compact-Kassette einlegen, da sonst nachfolgende Programme auf der Programm-Kassette überschrieben werden.

Beispiel:

```
100 REM LESEN DER ABGELEGTEN ZEICHEN
105 PRINT CHR$(7)"DATENKASSETTE EINLEGEN!"
110 OPENIN "!ASCII"
120 INPUT #9,H$
130 PRINT H$
140 IF EOF THEN GOTO 180
150 INPUT #9,BUCHSTABEN$
160 PRINT BUCHSTABEN$;
170 GOTO 140
180 PRINT "ENDE"
```

Hinweis:

Das Beispiel für das Ablegen der Daten finden Sie unter OPENOUT.

Vergleichen Sie hierzu auch:

OPENOUT, OPENIN, CLOSEIN, CLOSEOUT, PRINT, INPUT, LINE INPUT, WRITE

OPENOUT

Format: OPENOUT "Dateiname"

Zweck:

Eröffnen eines Ausgabefiles für Kassette.

Anwendung:

OPENOUT eröffnet einen Datenfile auf der Kassette zum Abspeichern von Daten mit dem laufenden Programm und legt einen Datenpuffer an. Wird als erster Buchstabe des Dateinamens ein Ausrufezeichen "!" angegeben, so werden keine Bedienungshinweise für den Datacorder auf dem Monitor ausgegeben. Die Daten werden vor der Ausgabe auf den Datacorder in einen Puffer von 2k Bytes Länge eingelesen. Ist der Puffer gefüllt oder wird der Befehl CLOSEOUT gegeben, dann wird der Pufferinhalt blockweise abgespeichert. Beachten Sie hierbei, daß erst durch die CLOSEOUT-Anweisung die "Restdaten" aus dem Puffer abgespeichert werden.

Beachten Sie bei der Anwendung des Beispielprogramms, daß Sie dieses vorher mit der MERGE-Anweisung mit dem Beispielprogramm bei der OPENIN-Anweisung mischen können. Bevor Sie dieses Programm nutzen, sollten Sie unbedingt eine Daten-Compact-Kassette einlegen, da sonst nachfolgende Programme auf der Programm-Kassette überschrieben werden.

Vorsicht:

Die NEW-Anweisung löscht den ”Rest”-Pufferinhalt ohne vorhergehende Abspeicherung, wenn vorher nicht die CLOSEOUT-Anweisung benutzt wird.

Beispiel:

```
10 REM ABLEGEN DER ASCII-ZEICHEN AUF KASSETTE
15 PRINT CHR$(7)"DATENKASSETTE EINLEGEN!"
20 OPENOUT "ASCII"Kr22
30 PRINT #9,"DIES IST EIN TESTPROGRAMM"
40 FOR I=0 TO 200
50 PRINT #9,CHR$(32+I);
60 NEXT I
70 CLOSEOUT
80 END
```

Hinweis:

Anwendung des Ausrufezeichens und das Lesen der abgelegten Daten siehe OPENIN.

Vergleichen Sie hierzu auch:

OPENIN, CLOSEIN, CLOSEOUT, PRINT, INPUT, LINE INPUT, WRITE

OR

Logische Verknüpfung: OR

Zweck:

Ableiten von Entscheidungen für den Programmablauf.

Anwendung:

Die logische Verknüpfung a OR b ist genau dann wahr, wenn der Ausdruck "a" oder der Ausdruck "b" wahr ist, d.h., nur einer der beiden Ausdrücke muß wahr sein.

Wahrheitstafel:

WAHR = 1; FALSCH = 0

a	b	a OR b
0	0	0
1	0	1
0	1	1
1	1	1

Beispiel:

```
10 MODE 1:BORDER 18
20 PRINT "Geben Sie zwei Zahlen (0-9) ein!"
30 PRINT:INPUT "Erste Zahl ";K
40 PRINT:INPUT "Zweite Zahl ";L
50 IF K=5 OR K=7 OR L=7 OR L=5 THEN 80
60 PRINT:PRINT "SIE HABEN WEDER 5 NOCH 7 EINGEGEBEN"
70 END
80 PRINT:PRINT "SIE HABEN 5 ODER 7 EINGEGEBEN"
90 END
```

Vergleichen Sie hierzu auch: AND, NOT

ORIGIN

Format: ORIGIN ⟨X-Koordinate⟩,⟨Y-Koordinate⟩
[,⟨linker Rand⟩,⟨rechter Rand⟩,
⟨oberer Rand⟩,⟨unterer Rand⟩]

Zweck:

Setzen des Graphik-Cursors und Definieren eines Graphik-Bildschirmfensters.

Anwendung:

ORIGIN setzt den Graphikcursor und bezieht sich dabei auf das gesetzte Graphik-Bildschirmfenster, welches ebenfalls mit der ORIGIN-Anweisung geändert werden kann.

Nach dem Einschalten entspricht die Größe des Graphik-Bildschirmfensters der des Text-Bildschirmfensters.

Das Graphik-Bildschirmfenster ist unabhängig vom mit MODE gewählten Bildschirmformat.

Die Ursprungs-Koordinate 0,0 ist bei der ORIGIN-Anweisung für Graphik darstellungen die linke untere Ecke des Graphik-Bildschirmfensters.

Werden für Graphikbefehle Koordinaten außerhalb des vom ORIGIN-Anweisung gesetzten Bildschirmfensters angegeben, so wird ohne Fehlermeldung nur das dargestellt, was innerhalb des Bildschirmfensters liegt.

Hinweis:

Der durch ORIGIN gesetzte Graphikcursor wird weder durch CLG noch durch CLS auf die Koordinate 0,0 zurückgesetzt.

Beispiel:

```
10 MODE 1
15 BORDER 13
20 ORIGIN 100,100,20,400,340,10
25 CLG 13
30 DRAW 600,300,0
40 GOTO 40
```

Ergebnis:

In diesem Programm wird in Zeile 15 das Textbildschirmfenster mit BORDER hervorgehoben.

Die ORIGIN-Anweisung in Zeile 20 setzt dann ein neues Graphik-Bildschirmfenster mit:

dem linken Rand auf Position 20,

dem rechten Rand auf Position 400,

dem unteren Rand auf Position 10 und

dem oberen Rand auf Position 340.

In Zeile 25 wird dieses Fenster mit dem CLG-Befehl gelöscht und mit einer anderen Farbe ebenfalls hervorgehoben.

Außerdem wird durch ORIGIN der Graphikcursor auf die Position 100,100 gesetzt.

Dieser Startpunkt wird in Zeile 30 vom DRAW-Kommando benutzt, um – vergleicht man die angegebenen Koordinaten – bis zur Koordinate 600,300 eine Linie zu ziehen. Diese ist aber nur bis zum rechten Rand des Graphikbildschirmfensters sichtbar und daher verkürzt.

Vergleichen Sie hierzu auch: WINDOW, CLG, CLS

OUT

Format: OUT(⟨Port⟩,⟨Integerausdruck⟩)

Zweck:

Byteweises Ausgeben von Daten an einen Port.

Anwendung:

OUT(⟨Port⟩,⟨Integerausdruck⟩) gibt ein Byte vom Wert ⟨Integerausdruck⟩ an den ⟨Port⟩ aus. ⟨Integerausdruck⟩ muß zwischen 0 und 255 liegen, sonst erscheint die Fehlermeldung Inproper argument auf dem Monitor. Der Wert ⟨Port⟩ muß zwischen -32768 und 32767 liegen. Liegt ⟨Port⟩ außerhalb dieses Bereiches, dann wird die Fehlermeldung Overflow ausgegeben.

Vergleichen Sie hierzu auch: INP, WAIT

PAPER

Format: PAPER [#Datenstrom,]⟨Code⟩

Zweck:

Setzen der Hintergrundfarbe eines Bildschirmfensters.

Anwendung:

Die PAPER-Anweisung setzt die Hintergrundfarbe bzw. das Bildschirmfenster auf die Farbe mit dem entsprechenden ⟨Code⟩ auf dem Farbmonitor. ⟨Code⟩ kann Werte zwischen "0" und "15" annehmen. Hierbei ist die vorher eingegebene MODE-Anweisung zu beachten. Nach Einschalten des ⟩⟩ CPC 464 ⟨⟨ benutzt PAPER den ⟨Code⟩ Null "0" im MODE 1, d.h., die Farbe des Bildschirmfensters ist gleich "1", was laut Farbcodetabelle blau entspricht. Im Format MODE 0 bewirken die ⟨Code⟩-Angaben "14" oder "15" einen Wechsel zwischen den beiden im Farbcode angegebenen Farben. PAPER bewirkt kein Löschen und keine sofortige Änderung der Hintergrundfarbe des Bildschirmfensters. Die Farbänderung erfolgt erst mit dem CLS-Kommando bzw. dann, wenn Zeichen geschrieben werden.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Werte größer 7 ergeben die Fehlermeldung `Improper argument`.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Beispiel 1:

```
10 MODE 1
20 PAPER 2
```

Ergebnis:

Die Hintergrundfarbe ist hellcyan.

Beispiel 2:

```
10 MODE 0
20 PAPER 4
```

Ergebnis:

Die Hintergrundfarbe ist hellweiß.

Beispiel 3:

```
10 MODE 0
20 PAPER 14
```

Ergebnis:

Die Hintergrundfarbe wechselt zwischen den Farben mit dem Farbcode "1" und "24", d.h., sie wechselt zwischen blau und hellgelb.

Vergleichen Sie hierzu auch:

SPEED INK, BORDER, PEN, INK, WINDOW

PEEK

Format: PEEK (⟨Adresse⟩)

Zweck:

Lesen des Inhalts einer Adresse.

Anwendung:

Mit PEEK (⟨Adresse⟩) wird der Inhalt der ⟨Adresse⟩ des Arbeitsspeichers gelesen. Der Wert für ⟨Adresse⟩ liegt zwischen &0000 und &FFFF. Die gelesenen Werte liegen zwischen 0 und 255.

Die PEEK- und POKE-Anweisungen kann man wirkungsvoll einsetzen zum:

- Speichern von Daten im Arbeitsspeicher
- Ablegen von Assemblerroutinen im Arbeitsspeicher
- Übergeben von Argumenten an Assemblerroutinen und
- Zurückgeben von Ergebnissen der Assemblerroutinen an BASIC-Programme.

Vergleichen Sie hierzu auch: POKE

Format: PEN [⟨Datenstrom⟩,]⟨Code⟩

Zweck:

Setzen der Zeichenfarbe.

Anwendung:

Die PEN-Anweisung setzt die Farbe der Zeichen auf die Farbe mit dem entsprechenden ⟨Code⟩. ⟨Code⟩ kann Werte zwischen "0" und "15" annehmen. Hierbei ist die vorher eingegebene MODE-Anweisung zu beachten.

Nach Einschalten des ⟩⟩CPC 464 ⟨⟨ benutzt PEN den ⟨Code⟩ eins "1" im MODE 1, d.h., die Farbe der Zeichen ist gleich "24", was laut Farbcodetabelle hellgelb entspricht. Im MODE 0-Format bewirken die ⟨Code⟩-Angaben "14" oder "15" einen Wechsel zwischen den beiden im Farbcode angegebenen Farben.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Werte gößer 7 führen zu der Fehlermeldung Improper argument.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Beispiel 1:

```
10 MODE 1
20 PEN 3
```

Ergebnis:

Die Farbe der Zeichen ist hellrot.

Beispiel 2:

```
10 MODE 0
20 PEN 4
```

Ergebnis:

Die Farbe der Zeichen ist hellweiß.

Beispiel 3:

```
10 MODE 0
20 PEN 14
```

Ergebnis:

Die Farbe der Zeichen wechselt zwischen den Farben mit dem Farbcode "1" und "24", d.h., sie wechselt zwischen blau und hellgelb.

Vergleichen Sie hierzu auch:

SPEED INK, BORDER, PAPER, INK, WINDOW

Format: `PI`

Zweck:

Ausgeben des Wertes 3.14159265

Anwendung:

Die Kreiszahl `PI` ist im BASIC-Interpreter implementiert und kann zur Berechnung mathematischer Funktionen direkt mit `PI` aufgerufen werden. `PI` wird häufig für mathematische oder graphische Programmteile benötigt.

Beispiel:

```
? PI,COS(PI)
```

Ergebnis:

```
3.14159265 -1
```

PLOT

Format: PLOT ⟨X-Koordinate⟩,⟨Y- Koordinate⟩[,⟨Code⟩]

Zweck:

Positionieren des Graphikcursors und Setzen eines Pixels in einer vorgegebenen Farbe.

Anwendung:

Die PLOT-Anweisung positioniert den nicht auf dem Monitor sichtbaren Graphik-Cursor und setzt einen Pixel (Graphikpunkt). Die Werte für die X- und Y-Koordinaten sind bezogen auf die Pixel-Koordinaten. Das Bildschirmfenster besteht aus 640 horizontalen und 400 vertikalen Bildpunkten, den sogenannten Pixels. Die untere linke Ecke des Bildschirmfensters besitzt die Koordinaten 0,0, die rechte obere die Koordinaten 639,399. Das PLOT-Kommando wird nicht von der Formatanweisung MODE beeinflußt, d.h., die eingegebenen Koordinaten behalten ihre Gültigkeit, nur die Pixelgröße und die Farbauswahl wird verändert.

Beispiel:

```
10 FOR I=0 TO 2 15 MODE I
20 BORDER 18
30 LOCATE 1,13
40 PRINT "MODE";I
50 PLOT 400,200,15
60 FOR K=2000 TO 0 STEP -1:NEXT K
70 NEXT I
80 END
```

Ergebnis:

Es wird die Pixelgröße in den unterschiedlichen Bildschirmformaten aufgezeigt.

Soll ein Pixel eine bestimmte Farbe annehmen, so ist der Aufruf wie nachstehend vorzunehmen:

```
10 MODE 0
20 PLOT 320,200,3
```

Der Pixel nimmt in diesem Fall die Farbe hellcyan an.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im `MODE 0` ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

`DRAW, DRAWR, PLOTR, MOVE, MOVER, ORIGIN, TEST, TESTR, XPOS, YPOS`

Format: PLOTR ⟨rel.X-Koordinate⟩,⟨rel.Y-Koordinate⟩[,⟨Code⟩]

Zweck:

Setzen eines Pixels in einer vorgegebenen Farbe relativ zur aktuellen Graphik-Cursorposition.

Anwendung:

PLOTR setzt relativ zur aktuellen Graphik-Cursorposition einen Pixel (Graphikpunkt). Die X-Koordinate des Punktes wird bestimmt durch die X-Koordinate der aktuellen Graphik-Cursorposition plus ⟨rel.X-Koordi nate⟩. Die Y-Koordinate berechnet sich entsprechend der X-Koordinate. Das PLOTR-Kommando wird nicht von der Formatanweisung MODE beeinflußt, d.h. die eingegebenen Koordinaten behalten ihre Gültigkeit, nur die Pixelgröße und die Farbauswahl wird verändert. PLOTR ist ähnlich dem DRAWR-Kommando, lediglich mit dem Unterschied, daß nur ein Pixel am Zielpunkt gesetzt wird.

Beispiel:

```
10 MODE 0
15 DATA 100, 20, -100, 20, -20, 100, -20, -100, -100,
-20, 100, -20, 20, -100, 20, 100
20 ORIGIN 350,200
30 FOR I=1 TO 8
40 READ X,Y
50 PLOTR X,Y,15
60 NEXT I
70 RESTORE
80 FOR K=0 TO 1500: NEXT K 90 FOR J=1 TO 8
100 READ X,Y
110 DRAWR X,Y
120 NEXT J
130 GOTO 130
```

Ergebnis:

Dieses Programm setzt zunächst die Eckpunkte (Pixel) eines Sterns und verbindet diese dann mit dem Kommando **DRAWR**. Die ⟨Code⟩-Angabe bewirkt ein Blinken der Pixel und danach des Sterns in den Farben rosa und himmelblau.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im **MODE 0** ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

PLOT, DRAW, DRAWR, MOVE, MOVER, ORIGIN, TEST, TESTR, XPOS, YPOS

POKE

Format: POKE ⟨Adresse⟩,⟨Wert⟩

Zweck:

Setzen eines bestimmten Bytes an einem definierten Speicherplatz.

Anwendung:

Mit POKE wird die ⟨Adresse⟩ mit dem ⟨Wert⟩, der zwischen 0 und 255 liegen muß, gefüllt. Die ⟨Adresse⟩ kann zwischen 0 und 65535 liegen. POKE-Anweisungen sind mit Vorsicht anzuwenden, da sie zu unerwarteten Effekten führen und ab und zu nicht mehr auf einfache Art und Weise behoben werden können, wie z.B. ein "Absturz des Rechners".

Vergleichen Sie hierzu auch: PEEK, HIMEM, MEMORY

Format: POS [#⟨Datenstrom⟩]

Zweck:

Ermitteln der aktuellen horizontalen Position des Textcursors.

Anwendung:

POS gibt die aktuelle horizontale Position des Textcursors in dem mit dem ⟨Datenstrom⟩ angegebenen Bildschirmfenster an. Die Ursprungskoordinate des Textbildschirmfensters entspricht hierbei der Position 1. Wird mit dem ⟨Datenstrom⟩ der Drucker angesprochen, so wird die Druckkopfposition ausgegeben, wobei diese berechnet wird und nur Zeichen mit einem ASCII-Code größer als 31 gezählt werden, d.h., Steuerzeichen werden nicht einberechnet.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7
#8	Drucker

Beispiel:

```
10 MODE 1
20 WINDOW #2,5,40,10,20
30 PAPER #2,2
40 INK 2,0
50 CLS #2
60 PRINT #2,"Das Textende ist auf Position "POS(#2)
70 END
```

Vergleichen Sie hierzu auch: VPOS

Format:

PRINT [#⟨Datenstrom⟩,][⟨Liste der Ausdrücke⟩]

bzw.

? [#⟨Datenstrom⟩,][⟨Liste der Ausdrücke⟩]

Zweck:

Ausgeben von Zeichen auf den vorgegebenen Ausgabeort.

Anwendung:

PRINT und Fragezeichen sind bei der Eingabe von Programmen und im Kommandomodus gleichwertig. Fragezeichen ist hierbei die kürzere An weisungsform und wird von BASIC in PRINT automatisch umgewandelt.

Die PRINT-Anweisung schreibt Texte und Zahlen in einem Bildschirmfenster. Wird der ⟨Datenstrom⟩ nicht näher spezifiziert, so wird zur Ausgabe das Bildschirmfenster, in dem sich der Textcursor befindet, benutzt.

Fehlt die Angabe ⟨Liste der Ausdrücke⟩, so wird eine Leerzeile ausgegeben.

Die Ausdrücke in der ⟨Liste der Ausdrücke⟩ sind entweder mit Kommata, Semikola oder Leerzeichen zu trennen. Semikolon und Leerzeichen besitzen die gleiche Wirkung.

Trennzeichen	Wirkung
Kommata ","	Setzt die Ausdrücke auf die entsprechende Bildschirmzone – siehe hierzu ZONE.
Semikola ";"	Setzt die Ausdrücke aneinander.
Leerzeichen	Setzt die Ausdrücke aneinander.

Die Ausgabe von numerischen Werten erfolgt immer mit nachgestelltem Leerzeichen. Positive Werte erhalten anstelle des Vorzeichens ein Leerzeichen, negative Werte werden dagegen mit ihrem Vorzeichen geschrieben.

Wird die ⟨Liste der Ausdrücke⟩ mit einem Semikolon oder Komma abgeschlossen, so erfolgt die nächste Bildschirmausgabe in der gleichen Zeile, wenn dafür die Zeilenlänge ausreichend ist. Bleibt der Abschluß der ⟨Liste der Ausdrücke⟩ offen, so wird von BASIC automatisch der Textcursor an den linken Rand der folgenden Zeile des entsprechenden Bildschirmfensters gesetzt.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7
#8	Drucker
#9	Datacorder

Werte größer 9 führen zu der Fehlermeldung `Improper argument`.

Beispiel 1:

```
5  MODE 1
10 INK 0,2:INK 2,4:INK 3,0:INK 1,26
20 PAPER 3
30 CLS
40 BORDER 9
50 WINDOW #1,7,33,4,18
60 PAPER #1,2
70 CLS #1
80 LOCATE #1,1,3
90  INPUT  #1,"NACHNAME  UND  PERSONALNUMMER  EINGEBEN
",NAME$
100 LOCATE #1,10,5:INPUT #1,"",ZAHL1
110 LOCATE #1,5,8
120 PRINT #1,"NACHNAME: "NAME$
130 LOCATE #1,5,10
140 PRINT #1,"PERSONALNUMMER: "ZAHL1
150 WINDOW #2,10,31,20,23
160 PAPER #2,4
170 CLS #2
180 LOCATE #2,2,3
190 PRINT #2,"ABTEILUNG SICHERHEIT"
200 LOCATE 5,2
210 PRINT "PROGRAMM MIT ESC-TASTE ABBRECHEN"
220 GOTO 220
```

Die PRINT-Anweisungen in den Zeilen 120,140 und 190 besitzen eine ⟨Datenstrom⟩-Angabe, diese bewirkt das Schreiben in bestimmten Bildschirmfenstern.

Die PRINT-Anweisung in Zeile 210 schreibt in das Bildschirmfenster "0", da hier keine ⟨Datenstrom⟩-Angabe vorgegeben wurde.

Beispiel 2:

```
120 OPENOUT "NAMEN"
130 PRINT #9,A$
140 CLOSEOUT
```

In Beispiel 2 werden Daten auf den Datacorder mit der PRINT-Anweisung in Zeile 130 geschrieben.

Beispiel 3:

```
165 PRINT #8,USING "##.#####";PI
```

In Beispiel 3 wird in dem von der Anweisung PRINT USING 3 vorgegebenen Format PI auf dem angeschlossenen Drucker ausgegeben.

Vergleichen Sie hierzu auch:

PRINT USING, WRITE, TAB, SPC, SPACE$, ZONE

Format: `PRINT [#`⟨Datenstrom⟩`,] USING` ⟨Format⟩`;`⟨Datenliste⟩

Zweck:

Ausgeben von formatierten Daten auf dem vorgegebenen Ausgabeort.

Anwendung:

Mit der `PRINT-USING`-Anweisung können Daten formatiert auf dem mit ⟨Datenstrom⟩ angegebenen Bildschirmfenster ausgegeben werden. Wird die Angabe ⟨Datenstrom⟩ nicht vorgegeben, so benutzt BASIC das vom Textcursor benutzte Bildschirmfenster.

Formatangaben

Symbol	Formatausführung
#	Gibt eine Ziffer einer Zahl aus. Das Ausgabeformat entspricht der Anzahl der Symbole. Wird diese kleiner als die Zahl, so richtet sich die Ausgabe nach der Zahl und setzt ein Prozentzeichen "%" als Warnung vor die Zahl.
.	Bestimmt die Position des Dezimalpunktes bei einer Zahl.
+	Gibt das Vorzeichen Plus "+" bei positiven Zahlen aus, bei negativen das entsprechende negative Vorzeichen.
**	Füllt bei einer kleineren Zahl die führenden Leerstellen mit Sternen "*" auf und entspricht in der Formatausführung dem Symbol "#".
$$	Es wird ein Dollarsymbol vor der ersten Ziffer ausgegeben und entspricht in der Formatausführung dem Symbol "#".

**$ Füllt die führenden Leerstellen mit Sternen auf und gibt ein Dollarsymbol vor der ersten Ziffer aus.

^^^^ Gibt eine Zahl im Exponentialformat aus.

, Fügt alle drei Stellen links vom Dezimalpunkt ein Komma ein.

! Gibt bei einer Zeichenkette nur das erste Zeichen aus.

& Gibt ein variables Format an.

⟨space⟩ Gibt entsprechend viele Leerzeichen aus.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7
#8	Drucker
#9	Datacorder

Vergleichen Sie hierzu auch: `PRINT, TAB, SPC, SPACE$`

Format: RAD

Zweck:

Umschalten auf Bogenmaß.

Anwendung:

RAD bewirkt bei Aufruf einer mathematischen Funktion, daß die Berechnung im Bogenmaß (Radiant) vorgenommen wird.

Vergleichen Sie hierzu auch: DEG, SIN, COS, TAN, ATN

RANDOMIZE

Format: RANDOMIZE [⟨Startwert⟩]

Zweck:

Setzen des Startwertes für den Zufallszahlengenerator.

Anwendung:

RANDOMIZE(X) bestimmt den Startpunkt in der Zufallszahlenreihe. Wird kein Wert für X angegeben, so erscheint auf dem Monitor die Abfrage Random number seed ? . Der Benutzer gibt hier eine Zahl ein, die der Rechner als neuen Startpunkt seiner internen Zufallszahlenreihe verwendet. Diese Abfrage erscheint nicht, wenn ein Wert für X vorgegeben wird.

Beispiel:

```
10 MODE 1
20 RANDOMIZE 341

30 PRINT RND RND(0) RND
```

Ergebnis:

```
0.951628158 0.951628158 0.934862386
```

Vergleichen Sie hierzu auch: RND

Format: READ ⟨Liste der Variaben⟩

Zweck:

Lesen von Daten aus DATA-Anweisungen bzw. DATA-Zeilen.

Anwendung:

READ liest genausoviele Werte aus DATA-Anweisungen, wie Variablen in der Liste aufgeführt sind und ordnet die Werte den Variablen zu. Während die INPUT-Anweisung dazu gedacht ist, Größen einzugeben, die von Fall zu Fall verschieden sind, wird die READ-Anweisung verwendet, um solche Werte an Variablen zu übergeben, die bei jedem Programmablauf stets in gleicher Weise benötigt werden. Die Daten werden deshalb bereits im Programm in besonderen DATA-Zeilen in genau der Reihenfolge bereitgestellt, in der sie später beim Ablauf des Programms benötigt werden.

Beispiel:

```
10 MODE 1
20 ON ERROR GOTO 130
30 LOCATE 5,5:PRINT "Name" TAB(28)"DM"
40 LOCATE 5,6:PRINT STRING$(30,"")
50 I=0
60 READ A$
70 LOCATE 5,7+I:PRINT A$;TAB(18);
80 INPUT "          ",DM
90 SUMME=SUMME+DM
100 I=I+1
110 GOTO 60
120 DATA Schulze,Meier,Schmitz
130 LOCATE 5,10:PRINT STRING$(30,"")
140 LOCATE 5,11:PRINT "Summe:" TAB(26);SUMME
150 LOCATE 5,15:PRINT "Programmende!"
160 END
```

Ergebnis:

Name	DM
Schulze	12.30
Meier	25.40
Schmitz	89.35
Summe:	127.05

Programmende!

Vergleichen Sie hierzu auch: `DATA, RESTORE`

RELEASE

Format: RELEASE ⟨Kanal⟩

Zweck:

Aufheben eines Wartezustandes bei dem vorgegebenen Tonkanal.

Anwendung:

Die RELEASE-Anweisung hebt Haltepunkte, die von der SOUND-Anweisung gesetzt wurden, wieder auf. Der Wert von ⟨Kanal⟩ ist dezimal, aber bitbezogen anzugeben, d.h., der Wert 1 entspricht dem Kanal A, 2 entspricht dem Kanal B und 4 dem Kanal C.

Vergleichen Sie hierzu auch: SOUND, SQ, ON SQ GOSUB

REM

Format: REM ⟨Kommentar⟩

Zweck:

Kennzeichnen einer Kommentarzeile.

Anwendung:

REM kennzeichnet Kommentarzeilen in einem Programm. Diese sogenannten REM-Zeilen dienen der Programmdokumentation und haben keinen Einfluß auf den Programmablauf. Eine REM-Anweisung kann an jeder beliebigen Stelle im Programm eingefügt werden. Folgende Punkte sind zu beachten:

1. REM-Anweisungen können mit GOTO- und GOSUB-Anweisungen angesprungen werden. Die Ausführung des Programms wird erst bei der nächsten ausführbaren Anweisung fortgesetzt. Für den Programmablauf ist aber günstiger, die nächste ausführbare Zeile direkt anzuspringen.

2. Unter diesem BASIC-Interpreter können REM-Anweisungen auch mit einem Apostroph " ´ " eingeleitet werden.

3. Eine REM-Anweisung kann auch an das Ende einer Programmzeile angehängt werden, da BASIC alles nach einem Apostroph als Kommentar auswertet.

4. Die REM-Anweisung darf nicht innerhalb einer Programmzeile eingefügt werden, da der rechte Teil der Zeile dann keine Programmfunktion mehr besitzt.

1.

```
10 REM ***** INDEX - Datei ***** SPEICHERN
```

2.

```
10 ' ***** Programm SOUND 3
```

3.

```
100 PRINT BLATT,SEITE,ZEILE,SPALTE:' Ausgabe
```

4. Ohne den Doppelpunkt vor dem Apostroph, was zulässig ist:

```
155 N=0: K=0'Zaehler loeschen
```

Beispiel für eine unzulässige REM-Anweisung:

```
55 I=12:'>ZEILE A$ SOLL GEDRUCKT WERDEN<:PRINT
#8,A$
```

REMAIN

Format: REMAIN ⟨Timer⟩

Zweck:

Abschalten des angegebenen ⟨Timer⟩ und Lesen der verbleibenden Restzeit.

Anwendung:

Die REMAIN-Anweisung setzt den angegebenen ⟨Timer⟩ außer Kraft und liest den Restwert der Zeitangabe. ⟨Timer⟩ besitzt die Werte 0 bis 3. Ist der Restwert 0, oder der angegebene ⟨Timer⟩ schon außer Kraft, so ist der durch die REMAIN-Anweisung ermittelte Wert 0.

Vergleichen Sie hierzu auch: AFTER, EVERY

Format:

RENUM [<Anfangszeilennummer>] [,<alte Zeilennummer>]
[,<Schrittweite>]

Zweck:

Um- bzw. Neunumerierung des Programms im Arbeitsspeicher.

Anwendung:

Die Anweisung RENUM dient dazu, die Zeilennummern eines Programms im Speicher einschließlich der darin enthaltenen Sprunganweisungen neu zu numerieren. Dies kann z.B. notwendig werden, wenn neue Programmzeilen einzufügen sind und die Zeilennummernabstände zu klein gewählt wurden.

Die Werte haben bei der RENUM-Anweisung folgende Bedeutung:

RENUM X,Y,Z

X gibt die neue Anfangszeilennummer an. Wird kein Wert für X angegeben, so wird mit Zeilennummer 10 begonnen.

Y gibt die alte Zeilennummer an, ab der eine Umnumerierung vorgenommen werden soll. Wird kein Wert für Y angegeben, so wird ab Programmanfang umnumeriert.

Z gibt die Schrittweite der Zeilennummern an. Wird Z nicht angegeben, so wird ein 10er Schritt ausgeführt.

Beispiele:

RENUM	Numeriert das gesamte Programm auf die neuen Zeilennummern 10, 20, 30, ... um.
RENUM 1000,,20	Numeriert ebenfalls das gesamte Programm neu bzw.
RENUM ,1000	Numeriert das alte Programm ab Zeilennummer 1000 um mit den Zeilennummern 1000, 1010, 1020, ...
RENUM 100,60,5	Numeriert ab der alten Zeilennummer 60 um in 100, 105, 110, 115, ...

Fehlermeldungen:

⟨Improper argument⟩ wird ausgegeben, wenn zwei Zeilen mit der gleichen Zeilennummer entstehen würden.

Beispiel:

```
10 REM Beispiel
20 END

RENUM 10,20
Improper argument
Ready
```

Undefined line ⟨Zeilennummer⟩ in ⟨Zeilennummer⟩ wird ausgegeben, wenn bei der Umnumerierung eine Zeile mit einem Sprungbefehl zu einer Zeile gefunden wird, die nicht existiert.

RESTORE

Format: RESTORE [⟨Zeilennummer⟩]

Zweck:

Erneutes Lesen von DATA-Anweisungen.

Anwendung

Die RESTORE-Anweisung ermöglicht es, mehrmals die gleiche DATA-Zeile(n) mit einer READ-Anweisung zu lesen, d.h., sämtliche Datenelemente in den DATA-Zeilen werden wieder zur verfügungen gestellt.
Die Angabe ⟨Zeilennummer⟩ bezieht sich auf eine bestimmte DATA-Zeile mit der angegebenen ⟨Zeilennummer⟩, auf deren erstes Datenelement mit der nachfolgenden READ-Anweisung zugegriffen werden soll.
Wird diese Angabe nicht gemacht, so greift die READ-Anweisung immer auf die erste DATA-Zeile zu.

Beispiel

```
5  REM ** BEISPIELPROGRAMM GIBT FEHLERMELDUNG AUS ***
10 MODE 1
20 READ ZAHL1,ZAHL2,ZAHL3
30 DATA 50,80,90
40 PRINT ZAHL1,ZAHL2,ZAHL3
50 GOTO 20
```

Ergebnis:

```
50             80             90
1  DATA exhausted in 20
```

Die in Zeile 20 stehende READ-Anweisung konnte nicht ausgeführt werden, da keine Datenelemente mehr zur Verfügung standen.

Geben Sie nun zusätzlich die Zeile 45 ein:

```
45 RESTORE
```

und danach die RUN-Anweisung im Kommandomodus. Durch die eingefügte Zeile kann nun wieder erneut auf die DATA-Zeile zugegriffen werden, da der DATA-Zeiger mit RESTORE zurückgesetzt wird.

Das Programm ist übrigens nur noch mit der ESC-Taste abzubrechen.

Vergleichen Sie hierzu auch: READ, DATA

RESUME

Format:

`RESUME [⟨Zeilennummer⟩]`

bzw.

`RESUME NEXT`

Zweck:

Fortsetzen der Programmausführung nach einer Fehlerbehandlungsroutine.

Anwendung:

Die `RESUME`-Anweisung hat nur in Verbindung mit der Fehlerbehandlung in einer Fehlerroutine zusammen mit `ON ERROR GOTO` eine Bedeutung. Ist eine Fehlerbehandlung ausgeführt worden, so muß diese mit `RESUME` beendet werden, wobei eines der nachstehend aufgeführten Formate anzuwenden ist:

`RESUME`

Das Programm wird mit derjenigen Zeile fortgesetzt, in der der Fehler auftrat.

`RESUME NEXT`

Das Programm wird mit der Zeile fortgesetzt, die unmittelbar auf die fehlerverursachende Zeile folgt.

RESUME ⟨Zeilennummer⟩

> Die Weiterführung des Programms erfolgt bei der angegebenen Zeilennummer.

Wird bzw. kann ein Fehler nicht durch das Programm selbst behoben werden, so fügen Sie die Anweisung:

```
ON ERROR GOTO 0
```

ein.

Die Fehlerbehandlungsroutine wird dann verlassen und BASIC gibt anschließend eine eigene Fehlermeldung aus.

Wurde eine Fehlermeldung ausgegeben, so befindet sich BASIC wieder im Kommandomodus. Sie können nun den Fehler beheben und das Programm im Direktmodus mit der **RUN**- oder **GOTO**- Anweisung neu starten.

Vergleichen Sie hierzu auch: **ON ERROR GOTO**

RETURN

Format: RETURN

Zweck:

Rücksprung aus einem Unterprogramm.

Anwendung:

RETURN kennzeichnet das Ende in einem Unterprogramm. Wird diese Anweisung erreicht, wird ein Rücksprung in das Hauptprogramm ausgeführt. Dieser Rücksprung erfolgt zu der nächstfolgenden Anweisung, d.h., nach der Anweisung, die das Unterprogramm aufgerufen hatte. Programmieren Sie so, daß vor einem Unterprogramm immer eine END-, STOP- oder GOTO-Anweisung vorausgeht, damit es nicht zu der Fehlermeldung Unexpected RETURN in ⟨Zeilennummer⟩ kommt.

Beispiel:

```
5  REM *** Beispielprogramm mit Fehlermeldung ***
10 MODE 1
20 PRINT "Es wird ein Unterprogramm aufgerufen>"
30 PRINT
40 GOSUB 1000
50 PRINT "Im Hauptprogramm!"
60 PRINT
1000 REM ##### Unterprogramm #####
1010 PRINT "*** Im Unterprogramm ***"
1020 FOR I=1 TO 1200: NEXT I
1030 PRINT
1040 RETURN
```

Ergebnis:

```
Es wird ein Unterprogramm aufgerufen

***Im Unterprogramm ***

Im Hauptprogramm

***Im Unterprogramm ***

Unexpected RETURN in 1040
```

Die Fehlermeldung tritt nicht auf, wenn die Zeile 70 eingefügt wird:

70 END

Vergleichen Sie hierzu auch:

GOSUB, ON GOSUB, ON SQ GOSUB, AFTER GOSUB, EVERY GOSUB, ON BREAK GOSUB

RIGHT$

Format: RIGHT$(⟨String⟩,⟨Anzahl⟩)

Zweck:

Übertragen von rechtsstehenden Zeichen aus einer Zeichenkette.

Anwendung:

RIGHT$(A$,X) liefert eine Teilzeichenkette aus A$ mit der Länge X. Es werden aus der Zeichenkette A$ X Zeichen von rechts nach links in die Zielvariable links vom Gleichheitszeichen übertragen. Der Wert von X muß zwischen 0 und 255 liegen. Ist X größer als die Anzahl der Zeichen der Zeichenkette A$, so wird die Zeichenkette A$ vollständig übertragen. Ist X=0, so wird kein Zeichen von A$ übertragen. Die Fehlermeldung Improper argument ⟨Zeilennummer⟩ erscheint, wenn X negativ oder größer 255 ist.

Beispiel:

```
10 MODE 2
20 A$="DONAUDAMPFSCHIFFAHRTSGESELLSCHAFTSKAPITAENS
PATENT"
30 LOCATE 40,5:PRINT RIGHT$(A$,6)
40 B$=RIGHT$(A$,15)
50 LOCATE 31,7:PRINT B$
60 C$=RIGHT$(A$,44)
70 LOCATE 2,9:PRINT C$
80 D$="*****"
90 PRINT D$;RIGHT$(A$,0);D$
100 END
```

Ergebnis:

```
                                       PATENT

                              KAPITAENSPATENT

 DAMPFSCHIFFAHRTSGESELLSCHAFTSKAPITAENSPATENT
```

Vergleichen Sie hierzu auch: LEFT$, MID$, SPACE$, STRING$

Format: RND [(numerischer Ausdruck]

Zweck:

Ermitteln einer Zufallszahl.

Anwendung:

Die Funktion RND erzeugt eine "Pseudo-"Zufallszahl zwischen 0 und 1 aus einer intern vorgegebenen Zahlenreihe. Ist bei RND(X) der numerische Ausdruck X größer als Null, dann ergibt die Funktion RND jedes mal eine neue Zufallszahl. Gleiche Werte erhalten Sie, wenn für X Null angegeben wird.

Der Startpunkt in der Zufallszahlenreihe kann durch den Befehl RANDOMIZE vorbestimmt werden, damit auf Wunsch immer die gleiche Zahlenreihe ausgegeben wird.

Ein Beispiel zu RND(X) finden Sie unter RANDOMIZE.

Vergleichen Sie hierzu auch: RANDOMIZE

ROUND

Format: ROUND (⟨numerischer Ausdruck⟩[,Integerausdruck⟩])

Zweck:

Runden von einer vorgegebenen Zahl.

Anwendung:
ROUND(X,Y) rundet einen numerischen Ausdruck auf Y-Stellen nach dem Komma. Ist Y negativ, so bezeichnet Y die Stelle vor dem Komma.

Beispiel:

```
5  MODE 1
10 X=54321.1234
20 PRINT X
30 FOR I=5 TO -5 STEP -1
40 PRINT I,ROUND (X,I)
50 NEXT I
60 END
```

Ergebnis:

```
 5          4321.1234
 5          54321.1234
 4          54321.1234
 3          54321.123
 2          54321.12
 1          54321.1
 0          54321
-1          54320
-2          54300
-3          54000
-4          50000
-5          100000
```

Vergleichen Sie hierzu auch: **INT, FIX, CINT, ABS, CREAL**

RUN

Format: RUN "⟨Programmname⟩"

Zweck:

Laden und Starten eines Programms vom Datacorder.

Anwendung:

Dieser Befehl lädt ein Programm mit dem angegebenen ⟨Programmnamen⟩ von der Programmkassette in den Arbeitsspeicher des ⟩⟩ CPC 464 ⟨⟨ ein und startet danach unmittelbar das Programm ab der ersten Programmanweisung.

Der ⟨Programmname⟩ kann maximal 16 Tastaturzeichen aufweisen. Wird hierbei als erstes Zeichen des Programmnamens ein Ausrufezeichen " ! " geschrieben, so werden die Anweisungen der Rekorderbedienung und der gefundenen Programmnamen auf dem Monitor unterdrückt.

Beachten Sie bei dieser Anweisung, daß ein im Speicher vorhandenes Programm und alle Variablen gelöscht werden.

Beispiel 1:

RUN "Adressenprogramm"

gibt Bedienungshinweise auf dem Monitor aus, lädt das Programm "Adressenprogramm" in den Arbeitsspeicher, löscht ein vorhandenes Programm im Speicher und startet das Programm unmittelbar.

Beispiel 2:

```
RUN "!Telefon"
```

gibt keine Hinweise auf dem Monitor aus und entspricht sonst dem Beispiel 1.

Beispiel 3:

```
RUN " "
```

lädt das nächste auf der Kassette gefundene Programm in den Arbeitsspeicher und entspricht sonst dem Beispiel 1.

Beispiel 4:

```
RUN "!"
```

gibt keine Hinweise auf dem Monitor aus und entspricht sonst dem Beispiel 3.

Vergleichen Sie hierzu auch:

LOAD, CAT, MERGE, CHAIN MERGE, CHAIN

RUN

Format: RUN [⟨Zeilennummer⟩]

Zweck:

Starten von einem im Arbeitsspeicher befindlichen Programm.

Anwendung:

Mit der RUN-Anweisung ohne Angabe der ⟨Zeilennumer⟩ wird ein im Arbeitsspeicher des ⟩⟩ CPC 464 ⟨⟨ vorhandenes Programm gestartet. Wird die RUN-Anweisung mit Angabe der ⟨Zeilennummer⟩ benutzt, so wird ab der angegebenen ⟨Zeilennumer⟩ das Programm abgearbeitet. Vorherliegende Zeilen mit ihren Anweisungen werden dann im Programm selbst nicht mehr berücksichtigt, d.h., sie werden so behandelt als würden sie nicht existieren. Dieses ist zu beachten, wenn mit der RUN ⟨Zeilennummer⟩-Anweisung Programmteile getestet werden sollen. Hierbei kommt es dann häufig zu Fehlermeldungen, die darauf beruhen, daß mit der RUN-Anweisung in eine "FOR-NEXT-Schleife", "WHILE-WEND-Schleife" oder nach einer "GOSUB"-Anweisung in das Programm gesprungen wurde.

Außerdem muß man beachten, daß mit der RUN-Anweisung auch alle Variablen gelöscht werden. Ist dieses unerwünscht, so sollte mit der GOTO-Anweisung gearbeitet werden.

Beispiele:

`RUN`

startet ein Programm mit der ersten Zeile.

`RUN 132`

startet ein Programm ab der Zeile mit der Zeilennummer 132.

Vergleichen Sie hierzu auch: `RUN "⟨Programmname⟩"`, `GOTO`

SAVE

Format: SAVE "⟨Dateiname⟩"[,⟨Dateityp⟩][,⟨binäre Parameter⟩]

Zweck:

Abspeichern des im Arbeitsspeicher befindlichen Programms.

Anwendung:

SAVE speichert ein Programm mit dem angegebenen Programmnamen bzw. ⟨Dateinamen⟩ aus dem Arbeitsspeicher auf Kassette. Der ⟨Dateiname⟩ kann maximal 16 Tastaturzeichen aufweisen. Die Angabe des ⟨Dateinamens⟩, des Dateityps und der binären Parameter kann zur Programmabspeicherung entfallen.

Der Dateityp wird bestimmt durch:

A = ASCII-Abspeicherung des Programms
P = geschütztes Programm "LIST - Schutz"
B = Abspeicherung eines Speicherbereiches

Wird die Abspeicherungsform B genutzt, so ist zusätzlich die Angabe der Startadresse und der Länge des Speicherbereiches anzugeben. Wahlweise kann eine Einsprungadresse eines Maschinenprogramms nach folgendem Schema angegeben werden:

Beispiel 1:

SAVE "⟨Dateiname⟩", B, ⟨Startadresse⟩, ⟨Länge⟩, ⟨Einsprungadresse-⟩

Beispiel 2:

```
SAVE ""
```

BASIC speichert das Programm als unbenannte Datei ab und es erscheint der Hinweis `Saving Unnamed file block ⟩X⟨` auf dem Monitor.

Beispiel 3:

```
SAVE "!Textpro",P
```

BASIC speichert hierbei das Programm "Textpro" `LIST`-geschützt auf Kassette und unterdrückt Bedienungshinweise auf dem Monitor.

Vergleichen Sie hierzu auch:

`RUN "⟨Dateiname⟩", LOAD, CAT, MERGE, CHAIN MERGE, CHAIN`

SGN

Format: SGN(⟨numerischer Ausdruck⟩)

Zweck:

Ableiten von Entscheidungen nach dem Vorzeichen für den Programmablauf.

Anwendung:

SGN(X) ermittelt das Vorzeichen des numerischen Ausdrucks X. Dieses wird benutzt, um Programmverzweigungen entsprechend dem ermittelten Vorzeichen vorzunehmen.

Die SGN-Funktion ergibt für:

X ⟩ 0 den Wert 1
X = 0 den Wert 0
X ⟨ 0 den Wert -1

Die Verwendung von SGN in einem Programm wird deutlicher an einem kurzen Beispiel mit dem Befehl ON GOTO:

Gegeben	Berechneter Sprung	Zielzeile
X ist negativ	ON SGN(X) +2 GOTO 10,20,30	→ 10
X ist null	ON SGN(X) +2 GOTO 10,20,30	→ 20
X ist positiv	ON SGN(X) +2 GOTO 10,20,30	→ 30

Ein weiteres Programmbeispiel finden Sie bei ON GOTO.

Vergleichen Sie hierzu auch: ABS

Format: `SIN(`⟨numerischer Ausdruck⟩`)`

Zweck:

Berechnet den Sinus.

Anwendung:

`SIN` liefert den Sinus des ⟨numerischen Ausdrucks⟩ im Bogenmaß. Der ⟨numerischer Ausdruck⟩ ist dann ein Winkel in Grad, wenn zuvor mit dem Befehl `DEG` auf Grad umgestellt wurde. Mit `RAD` kann wieder auf Bogenmaß zurückgestellt werden.

Beispiel:

```
10 MODE 2
20 X=30
30 PRINT SIN(X),
40 DEG
50 PRINT SIN(X),
60 RAD
70 PRINT SIN(X)
80 END
```

Ergebnis:

```
-0.988031625          0.5          -0.988031625
```

Vergleichen Sie hierzu auch: `DEG, RAD, COS, TAN, ATN`

SOUND

Format:

SOUND ⟨Kanalstatus⟩, ⟨Ton-Periode⟩, [⟨Dauer⟩] [, ⟨Lautstärke⟩][, ⟨Lautstärken-Hüllkurve⟩] [, ⟨Ton-Hüllkurve⟩][, ⟨Geräusch-Periode⟩]

Zweck:

Erzeugen von Tönen bzw. Musik.

Anwendung:

Die SOUND-Anweisung erlaubt es, vom Kinderlied bis zum aktuellen "Hit" Musik bzw. einen Sound auf dem ⟩⟩ CPC 464 ⟨⟨ umzusetzen.

Die angegebenen Variablen sind alle Integerzahlen und besitzen folgende Bedeutung:

⟨Kanalstatus⟩:

Es stehen 3 SOUND Kanäle zur Verfügung, die über den Kanalstatus angesprochen werden können bzw. der gewünschte Kanalstatus über entsprechend gesetzte Bits gesetzt werden kann:

Bit	dez.	Anweisung
0	1	Sende Sound nach Kanal A bzw. 1
1	2	Sende Sound nach Kanal B bzw. 2
2	4	Sende Sound nach Kanal C bzw. 4
3	8	Rendezvous mit Kanal A
4	16	Rendezvous mit Kanal B
5	32	Rendezvous mit Kanal C
6	64	halte
7	128	MSB

Es sind somit drei unterschiedliche Klangfarben gleichzeitig spielbar.

Beispiel für die Eingabe der Angabe ⟨Kanalstatus⟩:

Sende Sound nach Kanal C und Rendezvous mit Kanal A und halte wird dezimal angegeben mit:

4+8+64=76

Die Werte für Kanalstatus liegen im Bereich von 1 bis 255.

⟨Tonperiode⟩:

Die Angabe ⟨Tonperiode⟩ kann Werte von 0 bis 4095 annehmen und ist eine Integerzahl. Sie gibt die Tonhöhe an und umfaßt acht Oktaven. Die Angabe 284 dezimal bezeichnet laut Definition das internationale "a" und die Angabe 478 das mittlere "c".

⟨Dauer⟩:

Die Viertelnote ist der Standard für die Notenlänge und ist beim Schneider-Computer ⟩⟩ CPC 464 ⟨⟨ mit der Angabe 60 (nach Prüfung) anzusetzen. Eine kürzere Note, z.B. eine 1/8 Note, hat demnach den Wert 30 und eine 4/4 Note den Wert 240.

Beachten Sie, daß die Dauer bzw. die Notenlänge von der Lautstärkenhüllkurve beeinflußt wird. Ist Schrittzahl multipliziert mit Pausenlänge kleiner der Dauer der Note, so wird die Länge der Note nicht gehalten.

⟨Lautstärke⟩:

Mit der Angabe ⟨Lautstärke⟩ wird die Anfangslautstärke einer Note festgelegt. Der Wertebereich liegt hierbei zwischen 0 und 7. Null ist die kleinste, sieben die größte der wählbaren Lautstärken, wenn die Note bzw. der Ton nicht mit Lautstärken-Hüllkurve modelliert wird. Wird eine Lautstärken-Hüllkurve arrangiert, so liegt der Bereich zwischen 0 und 15. Wird die Angabe nicht vorgenommen, so ist der Wert 4 vorgegeben.

⟨Lautstärken-Hüllkurve⟩:

Mit dieser Angabe wird die Lautstärken-Hüllkurve, die angesprochen werden soll, gesetzt. Die Werte können 0 bis 15 annehmen, d.h., maximal 16 Lautstärken-Hüllkurven sind denkbar. Diese werden mit der ENV-Anweisung definiert. Beachten Sie, daß die Lautstärken-Hüllkurve die Tonlänge beeinflußt.

⟨Tonhüllkurve⟩:

Die mit der ENT-Anweisung definierte ⟨Tonhüllkurve⟩ wird mit dieser Angabe angesprochen. Auch hier sind maximal 16 Tonhüllkurven definierbar und über diese Angabe abrufbar.

⟨Geräusch-Periode⟩:

Der Standardwert ist Null, d.h., dem Notenklang wird kein Geräusch unterlegt. Es kann immer nur eine ⟨Geräusch-Periode⟩ definiert werden, die auch auf allen Kanälen identisch ist. Hiermit sind SOUND-Kreationen gleich dem "Krieg der Sterne" nicht fern.

Ein kleines Anwendungsbeispiel, das Sie sicher auch kennen werden:

Beispiel:

```
10 FOR I=1 TO 30
20 READ NOTE,DAUER
25 ENV 1,5,1,2
30 SOUND 1,NOTE,DAUER,6,1
40 NEXT I
50 DATA 379, 60, 379, 60, 426, 60, 478, 120, 379, 60,
379, 60, 426, 60
60 DATA 478, 120, 379, 60, 358, 60, 319, 60, 319, 60,
358, 30, 379, 30
70 DATA 358, 60, 426, 60, 379, 60, 358, 60, 358, 60, 379,
30, 426, 30
80 DATA 379, 60, 379, 60, 379, 60, 358, 60, 319, 120,
379, 60, 379, 60
90 DATA 426, 60, 478, 120
```

Vergleichen Sie hierzu auch: ON SQ GOSUB, SQ, ENT, ENV, RELEASE

SPACE$

Format: SPACE$(⟨Anzahl⟩)

Zweck:

Ausgeben einer Anzahl Leerzeichen.

Anwendung:

SPACE$(⟨Anzahl⟩) liefert eine Zeichenkette von ⟨Anzahl⟩ Leerzeichen. ⟨Anzahl⟩ sollte eine ganze Zahl sein und darf nur im Bereich von 0 bis 255 liegen. Wird ⟨Anzahl⟩ negativ oder größer als 255 angegeben, so erscheint die Fehlermeldung Improper argument auf dem Monitor.

Beispiel:

```
10 MODE 2
20 BORDER 20
30 K=5
40 FOR I=-8 TO 8
50 LOCATE 5,K
60 PRINT SPACE$(I*I);CHR$(42)
70 K=K+1
80 NEXT I
90 END
```

Vergleichen Sie hierzu auch: SPC, PRINT, TAB

Format: PRINT[#⟨Datenstrom⟩,]SPC(⟨Anzahl⟩)

Zweck:

Erzeugen von Leerstellen.

Anwendung:

Die SPC-Anweisung wird immer in Verbindung mit der PRINT-Anweisung benutzt und erzeugt entsprechend der Angabe ⟨Anzahl⟩ Leerstellen. ⟨Anzahl⟩ kann sinnvolle Werte zwischen 0 und 159 annehmen. Über die Angabe ⟨Datenstrom⟩=8 kann vom Drucker die ⟨Anzahl⟩ Leerstellen aus gegeben werden. Ebenso können durch eine andere Angabe für ⟨Datenstrom⟩ Leerstellen auf den unterschiedlichen Bildschirmfenstern erzeugt werden.

Beispiel:

```
20 PRINT "SPC-Formatierung"SPC(26)"TEST"
```

Ergebnis:

Zunächst wird auf Bildschirmspalte 1 die erste PRINT-Anweisung ausgeführt, danach werden 26 Leerstellen erzeugt und der String TEST ausgegeben.

Vergleichen Sie hierzu auch: SPACE$, PRINT USING, TAB

SPEED INK

Format: SPEED INK ⟨Integerzahl⟩,⟩Integerzahl⟩

Zweck:

Definieren des Zeitintervalls zwischen zwei Farbwechsel.

Anwendung:

SPEED INK ermöglicht in einstellbaren Zeitintervallen einen Wechsel der Farben, die vorab mit der INK- bzw. BORDER-Anweisung definiert wurden. Der Wechsel zwischen den Farben erfolgt mit der SPEED INK- Anweisung in Schritten von 0,02 Sekunden.

Wird die SPEED INK-Anweisung genutzt, so gibt die erste Integerzahl den Zeitraum an, nach dem von der ersten zur zweiten Farbe gewechselt wird. Die zweite Integerzahl definiert den Zeitraum, nach dem wieder zur ersten Farbe gewechselt wird.

Beispiel:

```
10 MODE 0
20 BORDER 12,9
30 INK 0,9,12
35 INK 1,0,26
40 SPEED INK 100,50
60 END
```

Vergleichen Sie hierzu auch:

INK, BORDER, PEN, PAPER, TEST, TESTR

SPEED KEY

Format: SPEED KEY ⟨Startzeit⟩,⟨Zeitintervall⟩

Zweck:

Einstellen der Tasten-Repeat-Funktion.

Anwendung:

Die SPEED KEY-Anweisung ermöglicht es, die Repeatfunktion der Tastatur den eigenen Wünschen anzupassen. Die ⟨Startzeit⟩ gibt den Zeitpunkt an, von dem ab ein Zeichen wiederholt werden soll. Der ⟨Zeitintervall⟩ gibt den Zeitraum zwischen dem schon geschriebenen Zeichen und dem zu wiederholenden Zeichen an. Für ⟨Startzeit⟩ und ⟨Zeitintervall⟩ können Werte von 1 bis 255 eingegeben werden. Die vorgegebene Repeatfunktion besitzt umgerechnet bei der SPEED KEY-Anweisung den Wert: 10,10 .

Diese Angabe bezieht sich auf Einheiten von 0,02 Sekunden.

Ist der Wert größer "255" oder kleiner "0", so erscheint die Fehlermeldung: Improper argument.

Sehr kleine Werte für ⟨Startzeit⟩, z.B. "1", sind problematisch, da die Tastatur bzw. die Hardware eine bestimmte Trägheit besitzt, so daß auch ein kurzes Antippen einer Taste zur Ausgabe mehrerer gleicher Zeichen führt.

Beachten Sie dies, damit Programme noch durch die entsprechende Eingabe im Kommandomodus abgespeichert werden können.

Beispiel:

```
SPEED KEY 50,100
```

Vergleichen Sie hierzu auch: KEY DEF

SPEED WRITE

Format: SPEED WRITE X

Zweck:

Setzen der Schreib- und Lesegeschwindigkeit für den Datacorder.

Anwendung:

Die SPEED-WRITE-Anweisung gestattet es, Daten und Programme in zwei unterschiedlichen Geschwindigkeiten auf Kassette abzulegen.

SPEED WRITE 1 legt mit 2000 baud und

SPEED WRITE 0 legt mit 1000 baud auf Kassette ab.

Werden abgespeicherte Programme oder Daten wieder von Kassette eingelesen, so erkennt der 〉〉 CPC 464 〈〈 automatisch die vorgegebene Lesegeschwindigkeit.

Die nach dem Einschaltvorgang vorgegebene Geschwindigkeit ist SPEED WRITE 0, d.h., die Abspeichergeschwindigkeit beträgt hier 1000 Daten bits pro Sekunde.

Sollte irrtümlich für X ein negativer oder ein Wert größer "1" eingegeben sein, so erscheint die Fehlermeldung: Improper argument.

Vergleichen Sie hierzu auch: SAVE, LOAD

Format: SQ ⟨Kanalnummer⟩

Zweck:

Ermitteln des Tonkanalstatus.

Anwendung:

Die SQ-Anweisung wird dazu benutzt, die Werte der drei freien Plätze in der Warteschlange mit dem angegebenen Kanal zu prüfen.

Über die gesetzten Bits, bezogen auf den von SQ ermittelten Wert, können Rückschlüsse auf den Zustand des durch die ⟨Kanalnummer⟩ angegebenen Kanals gezogen werden.

Für ⟨Kanalnummer⟩ sind folgende Werte einzugeben:

Kanalnummer	Kanal
1	A
2	B
4	C

Bedeutung der einzelnen gesetzten Bits:

Bits	Bedeutung
0,1,2	Kennzeichnen die Anzahl der freien Eingänge in der Warteschlange.
3	Kennzeichnet den Rendezvous-Status mit Kanal A am Anfang der Warteschlange.
4	Kennzeichnet den Rendezvous-Status mit Kanal B am Anfang der Warteschlange.
5	Kennzeichnet den Rendezvous-Status mit Kanal C am Anfang der Warteschlange.
6	Ist gesetzt, wenn am Anfang der Warteschlange ein Haltepunkt vermerkt ist.
7	Ist gesetzt, wenn der Kanal aktiv ist.

Sind die Bits 1 und 7 gesetzt, so ist das Ergebnis von SQ gleich 2+128=130.

Beispiel:

```
10 MODE 1
20 FOR N=1 TO 20
30 PRINT N;
40 SOUND 1,N+12,100
50 WHILE SQ(1)127:WEND
60 NEXT N
```

Vergleichen Sie hierzu auch: `SOUND, ON SQ GOSUB`

Format: SQR(⟨numerischer Ausdruck⟩)

Zweck:

Berechnen der Quadratwurzel.

Anwendung:

Die SQR-Anweisung berechnet die Quadratwurzel aus dem angegebenen ⟨numerischen Ausdruck⟩. Der ⟨numerische Ausdruck⟩ darf nicht negativ werden.

Beispiel:

```
10 MODE 2
20 LOCATE 5,10
30 PRINT"Geben Sie zwei Seiten eines rechtwinkligen
Dreiecks ein!"
40 LOCATE 5,12
50 INPUT "Erste Seite: ";SEITE1
60 LOCATE 5,14
70 INPUT "Zweite Seite: ";SEITE2
80 SEITE3=SQR(SEITE1^2 + SEITE2^2)
90 LOCATE 5,16
100 PRINT "Die dritte Seite Ihres Dreiecks: ";SEITE3
110 PRINT:PRINT
120 END
```

Ergebnis:

```
Geben Sie zwei Seiten eines rechtwinkligen Dreiecks
ein!

Erste Seite: 2

Zweite Seite: 2

Die Dritte Seite Ihres Dreiecks: 2.82842713
```

Vergleichen Sie hierzu auch: EXP, LOG, LOG10

Format: `STOP`

Zweck:

Beenden einer Programmausführung.

Anwendung:

Die `STOP`-Anweisung darf in jeder Zeile eines Programms eingefügt sein. Das Programm selbst unterbricht hierdurch den Programmablauf, ohne offene Dateien zu schließen und gibt den Hinweis "Break in ⟨Zeilennummer⟩" auf dem Monitor aus.

BASIC befindet sich nun wieder im Kommando-Modus. Die Programmausführung kann mit den Kommandos `CONT` oder `GOTO` ⟨Zeilennummer⟩ fortgesetzt werden.

Vergleichen Sie hierzu auch: `END, CONT, GOTO, ON BREAK STOP`

STR$

Format: STR$(⟨Zahl⟩)

Zweck:

Umwandeln eines numerischen Wertes in eine Zeichenkette.

Anwendung:

STR$(⟨Zahl⟩) verwandelt den numerischen Wert ⟨Zahl⟩ in einen String und übergibt sie einer Stringvariablen, wobei die jeweiligen Vorzeichen "+" und "–" zum String gehören. Das positive Vorzeichen wird als "blank" (Leerzeichen) ausgegeben.

Beispiel:

```
5  MODE 1
10 A$=STR$(123)
20 B$=STR$(-123)
30 PRINT A$,B$
40 PRINT LEN(A$),:PRINT LEN(B$)
```

Ergebnis:

```
 123          -123
 4              4
```

Vergleichen Sie hierzu auch: VAL, PRINT, HEX$, BIN$

STRING$

Format:

STRING$(⟨Länge⟩,⟨ASCII-Wert⟩)

bzw.

STRING$(⟨Länge⟩,⟨String⟩)

Zweck:

Erzeugen einer definiert langen Zeichenkette bestehend aus einem bestimmten Zeichen.

Anwendung:

STRING$(⟨Länge⟩,⟨ASCII-Wert⟩) liefert eine Zeichenkette mit der angegebenen Länge. ⟨Länge⟩ muß eine ganze Zahl sein und darf Werte zwischen 0 und 255 annehmen. Die Zeichenkette besitzt ausschließlich die Zeichen mit dem ASCII-Wert, der vorgegeben werden muß.

STRING$(⟨Länge⟩,⟨String⟩) ist eine weitere Möglichkeit, diesen Befehl aufzurufen. ⟨Länge⟩ bestimmt auch hierbei die Länge der Zeichenkette und ⟨String⟩ gibt das entsprechende Zeichen an, aus der die Zeichenkette besteht.

Beispiel:

```
10 MODE 1
20 A$=STRING$(5,32)
30 B$=" Nochn Gedicht "
40 C$=STRING$(5,42)
50 PRINT A$;C$;B$;C$
60 LOCATE 6,4:PRINT STRING$(20,"")
70 PRINT TAB(6) "von Heinz Erhard"
80 LOCATE 6,7:PRINT STRING$(26,35)
90 END
```

Ergebnis:

```
     ***** Nochn Gedicht *****

     von Heinz Erhard
     ##########################
```

Vergleiche Sie hierzu auch: SPACE$

SYMBOL

Format: `SYMBOL` ASCII-Wert〉,〈Datenliste〉

Zweck:

Definieren von benutzereigenen Zeichen.

Anwendung:

Mit `SYMBOL` können neue Zeichen definiert werden. Die Zeichenmatrix besteht aus 8 mal 8 Punkten (Pixels), bzw. aus acht Spalten und acht Zeilen. In der 〈Datenliste〉 stehen zeilenweise die Daten für die Pixels, d.h. es müssen immer acht Daten für acht Zeilen in der Matrix angegeben werden.

Der 〈ASCII-Wert〉 gibt das Zeichen an, das umdefiniert werden soll. Auf dem Monitor bereits ausgegebene Zeichen werden von der SYMBOL-Anweisung nicht verändert, so daß mehrere Zeichensätze nebeneinander darstellbar sind.

Beispiel:

```
10 MODE 1
20 SYMBOL AFTER 50
30 SYMBOL 91,40,0,120,12,124,204,118,0
40 PRINT STRING$(30,91)"tsch"
50 END
```

Ergebnis:

Auf die Taste ”eckige Klammer auf” wird ein ”” gelegt. Sowohl beim Drücken dieser Taste als auch beim Aufruf von CHR$(91) wird ein ”” geschrieben.

Vergleichen Sie hierzu auch: `SYMBOL AFTER`

SYMBOL AFTER

Format: SYMBOL AFTER ⟨ASCII-Wert⟩

Zweck:

Setzen des ASCII-Werte-Bereiches für die Umdefinition von Zeichen.

Anwendung:

Mit SYMBOL AFTER wird die Definition von neuen Zeichen ermöglicht. Es können Zeichen ab dem angegebenen ⟨ASCII-Wert⟩ neu definiert werden. Darunterliegende Werte können nicht mit SYMBOL umdefiniert werden. Nach dem Einschalten des ⟩⟩ CPC 464 ⟨⟨ liegt dieser ⟨ASCII-Wert⟩ bei 240.

Diese Anweisung sorgt für den notwendigen Pufferspeicher für die neu definierten Zeichen. Dieser Pufferspeicher kann eine maximale Größe von 2k Bytes annehmen. Sollen keine Zeichen neu definiert werden, so kann mit der "SYMBOL AFTER 256"-Anweisung zusätzlicher Speicherplatz geschaffen werden. Sind neue Zeichen definiert worden, so löscht eine erneute SYMBOL AFTER-Anweisung dieses Zeichen.

Ein Anwendungsbeispiel finden Sie bei der SYMBOL-Anweisung.

Vergleichen Sie hierzu auch: SYMBOL

TAB

Format: PRINT[#⟨Datenstrom⟩,]TAB(⟨Position⟩)

Zweck:

Positionieren des Textcursors.

Anwendung:

Die TAB-Anweisung wird immer in Verbindung mit der PRINT-Anweisung benutzt und positioniert entsprechend der Angabe ⟨Position⟩ den Textcursor. ⟨Position⟩ kann sinnvolle Werte zwischen 1 und 160 annehmen. Steht der Textcursor schon rechts von der mit der TAB-Anweisung angegebenen Position, so wird diese unwirksam und der Textcursor an die entsprechende Stelle der nächsten Zeile gesetzt. Über die Angabe ⟨Datenstrom⟩=8 kann die Druckkopfposition des Druckers durch die Angabe der ⟨Position⟩ gesetzt werden. Ebenso können durch eine andere Angabe für ⟨Datenstrom⟩ der Textcursor auf die unterschiedlichen Bildschirmfenster gesetzt werden.

Beispiel:

```
120 PRINT "TAB-Formatierung⟩"TAB (26)"TEST"
```

Ergebnis:

Zunächst wird auf Bildschirmspalte 1 die erste PRINT-Anweisung ausgeführt, danach auf Bildschirmspalte 26 der Textcursor positioniert und der String TEST ausgegeben.

Vergleichen Sie hierzu auch: SPACE$, PRINT USING, SPC

TAG / TAGOFF

Format: `TAG [#⟨Datenstrom⟩]` bzw. `TAGOFF [#⟨Datenstrom⟩]`

Zweck:

Mischen von Graphik mit Texten bzw. Ausgeben von Texten auf der Graphik-Cursorposition.

Anwendung:

Die `TAG`-Anweisung erlaubt es, Texte, die normalerweise auf der Textcursorposition geschrieben werden, umzuadressieren, um diese auf die Graphikcursorposition zu schreiben.

Hierdurch können Texte und Symbole, mit Graphik gemischt, in den Bildschirmfenstern dargestellt werden.

Die Graphikcursorposition ist bezogen auf die Ursprungskoordinate 0,0 und entspricht der linken unteren Ecke des Bildschirmfensters.

Die Graphikcursorposition kann mit der `MOVE`-Anweisung positioniert werden. Siehe hierzu Zeile 80.

Zu beachten ist hierbei, daß abschließend nach der Stringvariablen ein Semikolon gesetzt wird, da es sonst zu einer unkontrollierten Ausgabe von Graphikzeichen kommt. Siehe hierzu Zeile 100 des Programmbeispiels.

Die `TAGOFF`-Anweisung annulliert die `TAG`-Anweisung. Der Text wird nun an der vorherigen Textcursorposition geschrieben, d.h., genau an der aktuellen Stelle, bevor die `TAG`-Anweisung aufgerufen wurde. Siehe hierzu Zeile 120 im Beispielprogramm.

Die Angabe #⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Ist der Wert für #⟨Datenstrom⟩ größer als 7, so wird die Fehlermeldung `Improper argument` ausgegeben.

Beispiel:

```
10 BORDER 2
20 MODE 0
30 ORIGIN 1,1,150,500,60,180
40 CLG 5
50 FOR K=1 TO 5
60 READ A$
70 FOR I=1 TO 135
80 MOVE 140,60+I
90 TAG
100 PRINT A$;
110 TAGOFF
120 LOCATE 5,K+2:PRINT A$
130 NEXT I
140 NEXT K
150 DATA 64 K,COLOUR,MICRO,COMPUTER,⟨⟨CPC 464⟩⟩
160 MODE 1
170 END
```

Vergleichen Sie hierzu auch: `MOVE, LOCATE`

TAN

Format: TAN(⟨numerischer Ausdruck⟩)

Zweck:

Berechnen des Tangens eines numerischen Wertes.

Anwendung:

TAN liefert den Tangens des ⟨numerischen Ausdrucks⟩ im Bogenmaß. Der ⟨numerische Ausdruck⟩ ist dann ein Winkel in Grad, wenn zuvor mit dem Befehl DEG auf Grad umgestellt wurde.

Mit RAD kann wieder auf Bogenmaß zurückgestellt werden. Beim Überlauf im Rechenvorgang wird als Fehlermeldung Division by zero angezeigt.

Beispiel:

```
DEG:PRINT TAN(45)
```

Ergebnis:

```
1
```

Vergleichen Sie hierzu auch: DEG, RAD, SIN, COS, ATN

Format: TEST (⟨X-Koordinate,⟨Y-Koordinate⟩)

Zweck:

Ermitteln des Codes für die indirekte Farbbestimmung eines Punktes.

Anwendung:

Die Anweisung TEST liefert den benutzten Code an dem Punkt, der durch die ⟨X- und ⟨Y-Koordinate⟩ in der TEST-Anweisung angegeben wurde. Der CODE bezeichnet indirekt die Farbe des Zeichens bzw. Pixels an dieser abgefragten Position. Für die exakte Bestimmung der Farbe ist das mit der MODE-Anweisung gesetzte Bildschirmformat zu berücksichtigen.

Beispiel:

```
10 MODE 0
20 PLOT 50,50,7
30 PRINT TEST(50,50)
40 END
```

Ergebnis:

```
7
```

In Zeile 20 wird der Pixel mit der Koordinate 50,50 in Farbe hellmagenta gesetzt, weil ⟨Code⟩=7 in MODE 0 dem Farbcode 8 entspricht, was laut Farbcodetabelle hellmagenta ist.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im MODE 0 ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Vergleichen Sie hierzu auch:

`TESTR, MOVE, MOVR, PLOT, PLOTR, DRAW, DRAWR`

TESTR

Format: `TESTR (`⟨rel. X-Koordinate⟩`,`⟨rel. Y-Koordinate⟩`)`

Zweck:

Ermitteln des Codes zur indirekten Farbbestimmung des zur Graphik-Cursorposition relativ liegenden Punktes.

Anwendung:

Die Anweisung `TESTR` liefert den benutzten Code an dem Punkt, der durch die relativen Koordinatenangaben in der TESTR-Anweisung angegeben wurde. Die Bildschirmformatangabe `MODE` ist hierbei zu berücksichtigen. Die absolute X-Koordinate dieses Punktes wird durch die X-Koordinate der aktuellen Graphikcursorposition + ⟨rel. X-Koordinate⟩ bestimmt. Die Y-Koordinate berechnet sich entsprechend der X-Koordinate.

Code-Tabelle

⟨Code⟩		MODE 0	MODE 1	MODE 2
0	= Farbcode	1	1	1
1	= Farbcode	24	24	24
2	= Farbcode	20	20	1
3	= Farbcode	6	6	24
4	= Farbcode	26	1	1
5	= Farbcode	0	24	24
6	= Farbcode	2	20	1
7	= Farbcode	8	6	24
8	= Farbcode	10	1	1
9	= Farbcode	12	24	24
10	= Farbcode	14	20	1
11	= Farbcode	16	6	24
12	= Farbcode	18	1	1
13	= Farbcode	22	24	24
14*	= Farbcode	1⟩⟨24	20	1
15*	= Farbcode	16⟩⟨11	6	24

Die mit "*" gekennzeichneten Codes bewirken im `MODE 0` ein Wechseln zwischen den zwei Farben mit den angegebenen Farbcodes "⟩⟨".

Farbcode-Tabelle

Farbcode	Farbe	Farbcode	Farbe
0	schwarz	14	pastellblau
1	blau	15	orange
2	hellblau	16	rosa
3	rot	17	pastellmagenta
4	magenta	18	hellgrün
5	hellviolett	19	seegrün
6	hellrot	20	hellcyan
7	purpur	21	lindgrün
8	hellmagenta	22	pastellgrün
9	grün	23	pastellcyan
10	cyan	24	hellgelb
11	himmelblau	25	pastellgelb
12	gelb	26	hellweiß
13	weiß		

Ein Anwendungsbeispiel finden Sie bei der `TEST`-Anweisung.

Vergleichen Sie hierzu auch:

`TEST, MOVE, MOVER, PLOT, PLOTR, DRAW, DRAWR`

Format: `TIME`

Zweck:

Berechnen der abgelaufenen Zeit seit dem Einschalten.

Anwendung:

`TIME` gibt die Zeit in 1/300 sek an, die seit dem Einschalten des Computers vergangen ist. Die Berechnung dieser Zeitschritte erfolgt ohne Berücksichtigung der Kassettenrekorder-, Schreib- und -Lesezeiten.

Mit dem unten aufgeführten Programm können Sie den durch `TIME` ermittelten Wert in Stunden, Minuten und Sekunden umrechnen und ausgeben lassen.

Beispiel:

```
10 MODE 1
20 ZEIT=TIME
30 SEKUNDE=ZEIT/300
40 STUNDE=INT(SEKUNDE/3600)
50 SEKUNDE=SEKUNDE-STUNDE*3600
60 MINUTE=INT(SEKUNDE/60)
70 SEKUNDE=SEKUNDE-MINUTE*60
80 LOCATE 2,10:PRINT "Ihr Computer ist jetzt seit:"
90 LOCATE 1,12:PRINT STUNDE"Stunden
100 LOCATE 1,13:PRINT MINUTE"Minuten und"
110 LOCATE 1,14:PRINT SEKUNDE"Sekunden eingeschal-
tet."
120 PRINT:PRINT:END
```

TRON / TROFF

Format: TRON bzw. TROFF

Zweck:

Ein- bzw. Ausschalten des TRACE-Modus. Prüfen des Programmablaufs.

Anwendung:

TRON steht für TRACE ON und TROFF für TRACE OFF. TRON schaltet den TRACE-Modus ein, d.h., Programmabläufe werden verfolgt und auf dem Monitor dokumentiert. Das Programm durchläuft nacheinander Anweisung für Anweisung. Auf dem Monitor wird dabei die gerade bearbeitete Zeile durch die entsprechende Zeilennummer angezeigt. Diese Zeilennummern sind in eckige Klammern gesetzt, um sie z.B. von numerischen Ergebnis sen unterscheiden zu können.

Beachten Sie, daß die TRON-Anweisung nicht nur im Kommandomodus eingesetzt werden kann. Auch einer Teilprogrammanalyse kann sie dienen, indem TRON ab dem gewünschten Programmpunkt als Anweisung im Programm selbst steht.

Die TROFF-Anweisung ist die Umkehrung der TRON-Anweisung, d.h., sie schaltet den TRACE-Modus wieder aus. Benutzen Sie anstelle der TROFF-Anweisung die NEW-Anweisung, so schaltet auch diese Anweisung den TRACE-Modus wieder ab. Zu beachten ist hierbei aber, daß dann auch alle nicht abgespeicherten Daten bzw. Programmteile verloren sind.

Format: `UNT(X)`

Zweck:

Umwandeln einer 16-Bit-Integerzahl.

Anwendung:

`UNT(X)` wandelt eine positive 16-Bit-Integerzahl im Bereich von 0 bis 65535 in eine Integerzahl im Bereich von -32768 bis 32767 um.

Beispiel:

```
10 Mode 1
20 PRINT CHR$(10)CHR$(10) "Integerzahl UNT(Integer
zahl)"
30 PRINT STRING$(30,CHR$(&2D))
40 FOR I=1 TO 6
50 READ DATEN
60 DATA 0,1298,32767,32768,50001,65535
70 PRINT DATEN,UNT(DATEN)
80 NEXT I
90 PRINT CHR$(10):END
```

Ergebnis:

Integerzahl	UNT(Integerzahl)
0	0
1298	1298
32767	32767
32768	–32768
50001	–15535
65535	–1

Vergleichen Sie hierzu auch: INT, FIX, CINT

Format: UPPER$(⟨String⟩)

Zweck:

Umwandeln von Kleinbuchstaben in Großbuchstaben.

Anwendung:

UPPER$(A$) wandelt alle Kleinbuchstaben der Zeichenkette A$ in Großbuchstaben um.

Beispiel:

```
PRINT UPPER$("wichtige Mitteilung fuer den Benutzer!")
```

Ergebnis:

```
WICHTIGE MITTEILUNG FUER DEN BENUTZER!
```

Vergleichen Sie hierzu auch: LOWER$

VAL

Format: VAL(⟨String⟩)

Zweck:

Berechnen des Wertes einer als Zeichenkette angegebenen Zahl.

Anwendung:

VAL ist die Umkehrfunktion zu STR$. Sie liefert den numerischen Wert des ⟨Strings⟩. Das erste Zeichen des ⟨Strings⟩ muß eine Ziffer oder ein Vorzeichen "-, + ,&," sein, ansonsten hat VAL(⟨String⟩) den Wert Null "0".

Beispiel:

```
10 MODE 1
20 ZEILE=2:GOSUB 150
30 PRINT "STRING"
40 GOSUB 170:PRINT "VAL(STRING)"
50 PRINT STRING$(30,"")
60 FOR I=1 TO 5
70 READ A$
80 DATA &10,SPIEL 1,1. SPIEL,"        555",-13+2
90 ZEILE=ZEILE+2
100 GOSUB 150:PRINT A$
110 GOSUB 170:PRINT VAL(A$)
120 NEXT I
130 PRINT:PRINT
140 END
150 LOCATE 2,ZEILE
160 RETURN
170 LOCATE 20,ZEILE
180 RETURN
```

Ergebnis:

STRING	VAL(STRING)
&10	16
SPIEL 1	0
1. SPIEL	1
555	555
–13+2	–13

Vergleichen Sie hierzu auch: STR$

VERGLEICHS-OPERATOREN

Zweck:

Ableiten von Programmentscheidungen.

Anwendung:

Vergleichs-Operatoren ermöglichen einen Vergleich zwischen zwei numerischen Ausdrücken.

Das ermittelte Ergebnis dient der Ableitung von Programmentscheidungen, die z.B. für bedingte Programmsprünge genutzt werden können. Dieses Ergebnis kann bei allen Operatoren immer nur zwei Werte annehmen, 0 für "falsch" und "1" für wahr.

Das Gleichheitszeichen "=" wird auch bei der LET-Anweisung für die Wertzuweisung einer Variablen benutzt. Vergleichen Sie hierzu die LET-Anweisung. Beachten Sie bei der gleichzeitigen Anwendung von Vergleichs- und arithmetischen Operatoren, daß die letztgenannten bei der Abarbeitung Priorität besitzen.

Operator	Log. Ausdruck	Bedeutung
=	a = b	gleich
<>	a <> b	ungleich
<	a < b	kleiner als
>	a > b	größer als
<=	a <= b	kleiner gleich
>=	a >= b	größer gleich

Vergleichen Sie hierzu auch: **AND, OR, NOT**

VPOS

Format: `VPOS [#⟨Datenstrom⟩]`

Zweck:

Ermitteln der aktuellen vertikalen Text-Cursorposition.

Anwendung:

`VPOS` gibt die aktuelle vertikale Position des Text-Cursors in dem mit dem ⟨Datenstrom⟩ angegebenen Bildschirmfenster an. Die Ursprungskoordinate des Textbildschirmfensters entspricht hierbei der Position 1.

Die Angabe ⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Werte größer 7 ergeben die Fehlermeldung `Improper argument`.

Beispiel:

```
10 MODE 1
20 WINDOW #2,5,40,10,20
30 PAPER #2,2
40 INK 2,0
50 CLS #2
60 LOCATE #2,2,3
70 PRINT #2,"Der Textcursor ist in Zeile"VPOS(#2)
80 END
```

In Zeile 60 wird der Text-Cursor auf die 3. Zeile des in Zeile 20 vorgegebenen Textbildschirmfensters gesetzt.

Vergleichen Sie hierzu auch: POS

Format: `WAIT` ⟨Port⟩`,`⟨Und-Wert⟩`[,`⟨Oder-Wert⟩`]`

Zweck:

Abfragen eines bestimmten Bit-Status an einem Eingabe-Port.

Anwendung:

`WAIT` schiebt Anweisungen auf, bis an einem I/O Port ein bestimmtes Bitmuster anliegt. BASIC prüft hierbei das Bitmuster des I/O Ports in einer Abfrageschleife.

Das vom Port gelesene Byte wird durch ein "exclusiv oder" mit dem ⟨Oder-Wert⟩ und das Resultat dann mit dem ⟨Und-Wert⟩ verknüpft. Dieser Vorgang wiederholt sich bis das Ergebnis ungleich Null ist. BASIC wird so lange in der Abfrageschleife verbleiben, bis der erforderliche Wert erscheint, d.h., die `WAIT`-Anweisung unterbricht die Programmausführung bis zur Erkennung eines bestimmten Bit-Status am abgefragten Eingabe-Port.

Die Werte von ⟨Port⟩ liegen zwischen 0 und 255.

Vergleichen Sie hierzu auch: `INP, OUT`

WHILE ... WEND

Format: WHILE ⟨Bedingung⟩ WEND

Zweck:

Prüfen einer vorgegebenen Bedingung und davon abhängige Entscheidung für den Programmablauf.

Anwendung:

Die WHILE-WEND-Anweisung stellt eine Programmschleife dar, die solange von BASIC durchlaufen wird, wie eine bestimmte ⟨Bedingung⟩ wahr ist. Ist diese ⟨Bedingung⟩ nicht erfüllt bzw. ist diese "false", dann werden die Anweisungen, die zwischen WHILE und WEND stehen, von BASIC übersprungen und die nächste nach der WEND-Anweisung vorhandene Anweisung wird ausgeführt.

Zu beachten ist bei der WHILE-WEND-Anweisung, daß die ⟨Bedingung⟩ auch wahr bzw. "true" werden kann, da sonst die zwischen WHILE und WEND stehenden Anweisungen niemals ausgeführt werden.

Der Unterschied zur FOR-NEXT-Anweisung ist der, daß hierbei schon im Programm selbst die Schleifenanzahl festgelegt ist. Ist die Anzahl der Schleifendurchläufe vor dem Programmdurchlauf nicht bekannt, so bedient man sich der WHILE WEND-Anweisung.

Die Fehlermeldung "WEND missing in ⟨Zeilennummer⟩" wird ausgegeben, wenn für WHILE kein entsprechendes WEND gefunden wird.

Unexpected WEND in ⟨Zeilennummer⟩ erscheint, wenn für WEND kein WHILE existiert.

Beispiel:

```
10 REM **** WANN WERDE ICH MILLIONAER ****
20 CLS
30 LOCATE 5,2
40 PRINT STRING$(30,"*")
50 LOCATE 8,4
60 PRINT "Wann bin ich Millionaer?"
70 LOCATE 5,6
80 PRINT STRING$(30,"*")
90 LOCATE 8,8
100 INPUT "Verdienst im Monat";DM1
110 LOACATE 8,9
120 INPUT "Ausgaben im Monat";DM2
130 IF DM2>=DM1 THEN 140 ELSE 170
140 LOCATE 5,18
150 PRINT "Sie werden leider nie ein Millionaer"
160 END
170 SUMME=(DM1-DM2)*12
180 JAHR=0
185 REM ******** BEGINN DER WHILE WEND SCHLEIFE
*******
190 WHILE SPAREN<1000000
200 JAHR=JAHR+1
210 LOCATE 8,14
220 PRINT "Im";JAHR;". Jahr haben"
230 SPAREN=SUMME*JAHR
240 LOCATE 8,16
250 PRINT "Sie nun";SPAREN;"DM gespart."
260 WEND
265 REM ******* ENDE DER WHILE WEND SCHLEIFE
**********
```

```
270 LOCATE 8,18
280 PRINT "Sie sind nach";JAHR;"Jahren"
290 LOCATE 8,20
300 PRINT "Millionaer!"
310 IF JAHR>75 THEN 320 ELSE 350
320 LOCATE 8,22
330 PRINT "Erleben Sie dieses noch?"
340 END
350 LOCATE 8,2
2 360 PRINT "Herzlichen Glueckwunsch"
370 END
```

Die WHILE-WEND-Anweisung wird in diesem Programm solange durchlaufen, bis in Zeile 190 SPAREN größer 1000000 ist.

Vergleichen Sie hierzu auch: FOR ... NEXT

Format: `WIDTH` ⟨Spaltenzahl⟩

Zweck:

Begrenzen der Anzahl der Zeichen pro Zeile für die Druckerausgabe.

Anwendung:

Mit der `WIDTH`-Anweisung wird die Zeilenlänge des Druckens definiert. ⟨Spaltenzahl⟩ kann hierbei Werte zwischen 1 und 255 annehmen. Wird ein größerer Wert eingegeben, so erscheint die Fehlermeldung `Improper argument`. Diese Anweisung erlaubt es BASIC, beim Drucken notwendige Wagenrückläufe einzufügen.

Vergleichen Sie hierzu auch: `PRINT`, `POS`

WINDOW

Format:

WINDOW [⟨Datenstrom⟩,]⟨linker Rand⟩,
⟨rechter Rand⟩,
⟨oberer Rand,⟨unterer Rand⟩

Zweck:

Definieren eines Bildschirmfensters.

Anwendung:

Mit der WINDOW-Anweisung können bis zu 7 Text-Bildschirmfenster gesetzt werden. Die Bildschirmfenster können sich gegenseitig überlappen. Diese Bildschirmfenster werden mit der Angabe ⟨Datenstrom⟩ eindeutig definiert, so daß sie unter dieser Zahl spezifiziert aufgerufen werden können. Die Angaben zur Größe des Bildschirmfensters entsprechen der LOCATE-Anweisung.

Der Wertebereich der übrigen Größen ist:

⟨oberer Rand⟩	minimal	1
⟨unterer Rand⟩	maximal	25

Wird für ⟨Datenstrom⟩ ein Wert größer 7 eingegeben, so erscheint die Fehlermeldung `Improper argument`.

Beispiel:

```
10 MODE 1
20 FOR I=1 TO 3
30 WINDOW #I,I*5,I*12,I*5,I*8
40 PAPER #I,I
50 CLS #I
60 INK I+1,I,I+5
70 NEXT I
80 BORDER 9
90 GOTO 90
```

Ergebnis:

Sie erkennen auf Ihrem Farbmonitor 3 blinkende Bildschirmfenster.

Vergleichen Sie hierzu auch:

ORIGIN, WINDOW SWAP, LOCATE, LIST, PRINT, INPUT, LINE INPUT, CLS, PEN, PAPER, POS, TAG, TAGOFF, VPOS

WINDOW SWAP

Format: `WINDOW SWAP` ⟨Datenstrom⟩`,`⟨Datenstrom⟩

Zweck:

Tauschen der angegebenen Bildschirmfenster bzw. Datenströme.

Anwendung:

`WINDOW SWAP` tauscht den ersten ⟨Datenstrom⟩ mit dem zweiten. Der zweite angegebene ⟨Datenstrom⟩ ist dann das aktuelle Text-Bildschirmfenster mit der entsprechend definierten Fensternummer. Ist über die `WINDOW-SWAP`-Anweisung das Text-Bildschirmfenster gewechselt, so wird die nächste Anweisung in dem Text-Bildschirmfenster ausgeführt, indem sich der Textcursor befindet.
Ist der Textcursor in einem der angesprochenen Bildschirmfenster, so wird der Textcursor in das zweite mit ⟨Datenziel⟩ angegebene Bildschirmfenster gesetzt.

Die Angabe #⟩⟨⟨Datenstrom⟩ entspricht:

Datenstrom	Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7

Beispiel:

```
10 MODE 1
20 FOR I=1 TO 2
30 WINDOW #I,I*5,12*I,5*I,8*I
35 PAPER #I,I+1
36 CLS #I
40 NEXT I
50 PRINT #2,"FENSTERTEXT 1"
60 WINDOW SWAP 2,1
70 PRINT #2,"zweiter Text"
80 END
```

Ergebnis:

Obwohl bei der "`PRINT #`"-Anweisung in beiden Fällen das Bildschirmfenster mit Fensternummer "2" angesprochen wird, erscheint nach `WINDOW SWAP` der zweite Text im Bildschirmfenster "1".

Vergleichen Sie hierzu auch: `WINDOW`

WRITE

Format: WRITE [#⟨Datenziel⟩][,⟨Datenliste⟩]

Zweck:

Ausgeben von Daten an den angegebenen Ausgabeort.

Amwendung:

Die WRITE-Anweisung arbeitet ähnlich der PRINT-Anweisung. Der Unterschied ist der, daß die WRITE-Anweisung auszugebende Strings in Anführungszeichen setzt und Zahlen durch Kommata trennt. Dieses kann z.B. zur Ausgabe in Tabellen verwendet werden. Die WRITE-Anweisung erlaubt es weiterhin, Daten auf dem Datacorder abzulegen, sowie Daten über die parallele Schnittstelle auszugeben, um sie zu drucken.

Datenziel	Ein-/Ausgabeort
#0	Bildschirmfenster 0
#1	Bildschirmfenster 1
#.	...
#7	Bildschirmfenster 7
#8	Drucker
#9	Datacorder

Beispiel 1:

```
10 MODE 1
20 WINDOW #3,5,35,10,20
30 PAPER #3,2
40 INK 2,0
50 CLS #3
60 A=3:C=4.7E+26
70 WRITE #3,"CPC 464
80 PRINT #3
90 WRITE #3,A,PI,C
100 END
```

Ergebnis:

In Zeile 20 wird das Bildschirmfenster 3 mit der `WINDOW`-Anweisung definiert und in Zeile 70,80 und 90 wird in dieses Bildschirmfenster mit der WRITE-Anweisung geschrieben.

Beispiel 2:

```
10 MODE 1
20 OPENOUT "TEST"
30 S$="Dieser Text wurde auf dem Datacorder ausgegeben"
40 WRITE #9,S$
50 CLOSEOUT
55 CLEAR
60 OPENIN "TEST"
70 INPUT #9,Z$
80 PRINT CHR$(10)Z$
90 CLOSEIN
100 END
```

Beachten Sie bitte Zeile 55. Diese löscht alle Variablen, so daß hiermit nachgewiesen wird, daß der String `Z$` tatsächlich vom Datacorder eingelesen wird.

Ergebnis:

Dieses Programm speichert die Zeichenkettenvariable `S$` auf dem Datacorder ab. Danach erscheint der Hinweis `Press PLAY then any key`. Bevor Sie dieser Aufforderung nachkommen, spulen Sie das Band zurück, damit die abgelegten Daten wieder eingelesen werden können.

Vergleichen sie hierzu auch: `PRINT`

Logische Verknüpfung: `XOR`

Zweck:

Ableiten von Entscheidungen für den Programmablauf.

Anwendung:

Die logische Verknüpfung `a XOR b` ist genau dann wahr, wenn entweder nur der Ausdruck "a" oder nur der Ausdruck "b" wahr ist, d.h., nur einer der beiden Ausdrücke darf wahr sein.

Wahrheitstafel: WAHR = 1; FALSCH = 0

a	b	a XOR b
0	0	0
1	0	1
0	1	1
1	1	0

Beispiel:

```
10 FOR I=0 TO 1
20 FOR K=0 TO 1
30 PRINT "I="I" K="K"=> I XOR K ->"I XOR K
40 NEXT K
50 NEXT I
```

Vergleichen Sie hierzu auch: `AND, OR, NOT`

XPOS

Format: XPOS

Zweck:

Bestimmen der horizontalen Graphik-Cursorposition.

Anwendung:

Die XPOS-Anweisung bestimmt die horizontale Position (X-Koordinate) des Graphikcursors.

Beispiel:

```
10 MODE 1
20 PLOT 339,199
30 PRINT "Der Graphikcursor befindet sich"
40 PRINT "horizontal auf der Position"XPOS
50 PRINT "vertikal auf der Position"YPOS
60 LOCATE 20,13:PRINT CHR$(243)
70 END
```

Vergleichen Sie hierzu auch: YPOS, MOVE, MOVER, ORIGIN

Format: YPOS

Zweck:

Bestimmen der aktuellen vertikalen Graphik-Cursorposition.

Anwendung:

YPOS bestimmt die vertikale Position (Y-Koordinate) des Graphikcur sors.

Die Position des Graphikcursors wird durch dieses Kommando nicht verändert.

Beispiel:

```
10 MODE 1
15 BORDER 18
20 PRINT"Bitte geben Sie eine Koordinate ein"
30 INPUT"Die horizontale Position";X
40 INPUT"Die vertikale Position";Y
45 PRINT
50 DRAW X,Y,3
60 PRINT"Der Graphikcursor befindet sich"
70 PRINT "horizontal auf der Position"XPOS
80 PRINT "vertikal auf der Position"YPOS
90 END
```

Ergebnis:

Das Programm zeichnet eine Linie zum eingegebenen Punkt und bestimmt danach die aktuelle Position des Graphikcursors.

Vergleichen Sie hierzu auch: XPOS, MOVE, MOVER, ORIGIN

ZONE

Format: ZONE(⟨Integerzahl⟩)

Zweck:

Setzen der Bildschirmzonen.

Anwendung:

ZONE(X) setzt die vertikalen Bildschirmzonen, die die PRINT-Anweisung mit dem Komma als Formatierungsanweisung nutzt. Die Angabe von X bestimmt die Größe des Zwischenraums von zwei Ansprungpositionen und muß eine Integerzahl zwischen 1 und 255 sein. Die ZONE-Anweisung wird durch die NEW-, LOAD-, CHAIN- und RUN "⟨Programmname⟩"-Anweisung auf den ursprünglichen Wert 14 zurückgesetzt.

Beispiel:

```
10 MODE 2
20 PRINT "A","B","C"
30 ZONE 25
40 PRINT "A25","B25",C25"
50 END
```

Vergleichen Sie hierzu auch: PRINT, TAB, WIDTH

Einführung

Ein oder auch mehrere Fehler in einem Programm sind bei der Programmentwicklung fast eine Normalität. Die Fehler können einerseits nützlich, andererseits aber sehr lästig sein. Nützlich, weil aus jeder Fehlerbeseitigung neue Erkenntnisse abgeleitet werden können, die der eigenen Programmiererfahrung und somit den späteren Programmen zugute kommen.

Lästig, da der Zeitaufwand und die Enttäuschung groß sein kann, wenn ein kreiertes Programm nicht "läuft". Häufig handelt es sich hierbei nicht um logische Programmfehler, sondern im nachhinein so trivial erscheinende, daß sie vermeidbar gewesen wären.

Soft- und Hardware Fehler

Hier sind zunächst vier unterschiedliche Fehlerarten zu unterscheiden:

Fehlerarten:

* Syntax-Fehler ....: Software-Fehler
* Fehler in der Programmlogik ....: Software-Fehler
* Fehler durch den Programmbenutzer .: Software-Fehler
* Fehler der Technik ....: Hardware-Fehler

Ein Syntax-Fehler liegt immer dann vor, wenn eine BASIC-Format- Vorgabe nicht beachtet wird, z.B. wenn REED anstelle von READ eine WHILE-WEND-Schleife, deren Bedingung niemals erfüllt, d.h., "true" werden kann.

Diese zwei Fehlerarten werden meist während der Testphase des Programms erkannt und beseitigt. Das ist für die Lauffähigkeit eines Programms absolute Vorbedingung.

Fehler, die durch den Programmbenutzer bzw. bei der Nutzung des Programms auftreten, lassen auf eine zu kurze Testphase oder auf eine Nichtbeachtung aller Programmöglichkeiten schließen. Diese Fehlerart ist häufig anzutreffen, wenn z.B. Benutzereingaben über die Tastatur in das laufende Programm gefordert werden. Hier hilft nur eine sorgfältige Prüfung aller denkbaren Programmvorgänge.

Hardware-Fehler sind die problematischsten, da sie meistens nicht eigenhändig behoben werden können und zu einem unerwarteten Abbruch der Programmierarbeit führen.

Die einfachste, denkbare Fehlerursache ist hierbei ein Netzausfall oder das Durchbrennen einer Sicherung im Terminal oder in der Peripherie des Computers.

Ihr ⟩⟩ CPC 464 ⟨⟨ wertet die oben genannten "Software-Fehler" aus und meldet die meisten davon dem Programmierer durch Ausgabe einer Fehlermeldung.

Um Ihnen sämtliche originale Fehlermeldungen auch auf dem Monitor zu bieten, wurde ein Programm erstellt, das nacheinander alle diese Fehlermeldungen ausgibt.

Das BASIC-Programm finden Sie nachstehend aufgelistet:

Programm: "Fehlermeldungen des ⟩⟩ CPC 464 ⟨⟨"

```
10 REM PROGRAMM GIBT ALLE MOEGLICHEN
20 REM FEHLERMELDUNGEN VOM ⟩⟩ CPC 464 ⟨⟨
30 REM AUF DEM MONITOR AUS
35 REM ******************************************
40 CLS
50 MODE 1
60 LOCATE 5,2
70 PRINT "**** Alle Fehlermeldungen ****"
80 LOCATE 2,4
90 PRINT "Bitte Punkt im Ziffernblock druecken"
1 100 LOCATE 2,23
110 PRINT "Bei Fehler 2 bitte ESC-Taste druecken!"
120 I=0
130 KEY 138,"GOTO 140"+CHR$(13)
140 I=I+1
150 WINDOW #2,3,40,10,20
160 CLS #2
170 LOCATE #2,5,1
180 PRINT #2,"Fehler Nr.:"I
190 WINDOW SWAP 0,2
200 LOCATE 1,4
210 ERROR I
```

Sie konnten während des Programmablaufs erkennen, daß neben der Fehlermeldung auch eine Fehlernummer ausgegeben wurde. Diese Fehlernummer ist der entsprechenden Fehlermeldung fest zugeordnet, so daß z.B. Fehlernummer 6 die Fehlermeldung `Overflow in <Zeilennummer>` eindeutig kennzeichnet.

Unter BASIC kann die Fehlernummer in einer Fehlerbehandlungsroutine mit der `ERR`-Anweisung abgefragt werden. Die `ERL`-Anweisung ermittelt die Zeilennummer, in der der Fehler aufgetreten ist, so daß über beide Angaben ein Beheben des Fehlers durch das Programm selbst möglich wird.

Nachfolgend finden Sie aufgeführt:

* Alle Fehlermeldungen mit der dazugehörigen Fehlernummer

* Die Bedeutung bzw. die Übersetzung der Fehlermeldung

* Eine Erläuterung zur Entstehung eines Fehlers

* Programmbeispiele, die eine Fehlermeldung erzeugen

* Querverweise zu BASIC-Anweisungen

Falls in Ihrem Programm eine Fehlermeldung erscheint, so sollten Sie diesen Teil des Buches unbedingt nutzen, um den Fehler möglichst schnell lokalisieren und beheben zu können.

Fehlernummer: 1

Fehlermeldung: `Unexpected NEXT in` ⟨Zeilennummer⟩

Bedeutung: `NEXT` wurde ohne dazugehörige FOR-Anweisung gefunden.

Erläuterung:

Diese Fehlermeldung tritt auf, wenn die zur NEXT-Anweisung zugehörige Laufanweisung FOR nicht programmiert wurde. Eine weitere Ursache für diese Fehlermeldung kann ein Hineinspringen in die `FOR-NEXT`-Schleife sein, z.B. mit der GOTO-Anweisung.

Vergleichen Sie hierzu: `FOR ... NEXT`

Fehlernummer: 2

Fehlermeldung: `Syntax error in` ⟨Zeilennummer⟩

Bedeutung: BASIC-Schreibfehler; Verstoß gegen die BASIC-Sprachregeln.

Erläuterung:

Ein Syntaxfehler wird gemeldet, wenn eine BASIC-Rechtschreibregel nicht beachtet wird, z.B. `REED` anstelle von `READ`, oder wenn beim Setzen von Klammern die Zahl der sich öffnenden Klammern mit der Zahl der sich schließenden Klammern nicht übereinstimmt.

Vergleichen Sie hierzu: `EDIT`

Fehlernummer: 3

Fehlermeldung: `Unexpected RETURN in` ⟨Zeilennummer⟩

Bedeutung: Bei einer RETURN-Anweisung fehlt die dazugehörige GOSUB-Anweisung

Erläuterung:

Diese Fehlermeldung wird ausgegeben, wenn versucht wird, aus einem Unterprogramm ohne vorherigen ordnungsgemäßen Einsprung in dieses Unterprogramm zurückzukehren. Wie bei der `FOR-NEXT`-Schleife das Paar `FOR` und `NEXT` zusammengehören, so gehört auch das Paar "`GOSUB ... RETURN`" zusammen. Fehlt die `GOSUB`-Anweisung, wird die oben aufgeführte Fehlermeldung auf dem Bildschirm angezeigt. Der Fehler tritt ebenfalls auf, wenn von außen in ein Unterprogramm hineingesprungen wird, d.h. also, wenn die GOSUB-Anweisung nicht ordnungsgemäß durchlaufen wird.

Vergleichen Sie hierzu: `RETURN, GOSUB`

Fehlernummer: 4

Fehlermeldung: `DATA exhausted in` ⟨Zeilennummer⟩⟩

Bedeutung: Es fehlen Daten für die `DATA`-Anweisung.

Erläuterung:

Auch hier haben wir wieder zumindest ein Pärchen, bestehend aus einer oder mehreren `READ`-Anweisungen und einer oder mehreren `DATA`-Anweisungen, vor uns. "Gelesen" werden `DATA`-Anweisungen ausschließlich mit einer oder mehreren `READ`-Anweisungen. Die Fehlermeldung tritt dann auf, wenn alle vorhandenen `DATA`-Anweisungen bereits gelesen worden sind und ein erneuter Leseversuch unternommen wird. Mit anderen Worten: der Fehler wird angezeigt, wenn für eine `READ`-Anweisung keine `DATA`-Anweisung mehr vorhanden ist.

Vergleichen Sie hierzu: `READ, DATA`

Fehlernummer: 5

Fehlermeldung: `Improper argument in` ⟨Zeilennummer⟩

Bedeutung: Es wurde ein unerlaubtes Argument angegeben.

Erläuterung:

Diese Fehlermeldung kann die verschiedensten Ursachen haben. Sie bedeutet, daß der Wert eines Arguments einer Funktion nicht erlaubt ist oder bestimmte Parameter einer Anweisung falsch sind.

Beispiel:

```
PRINT SQR(-25)
Improper argument

MODE 5

Improper argument

WIDTH 400

Improper argument
```

Hinweis:

Wenn bei `ON ⟨M⟩ GOTO ...` oder auch bei `ON ⟨M⟩ GOSUB ...` der Wert des Ausdrucks gleich Null oder größer als die Anzahl der Listenelemente (aber ⟨=255) ist, so wird keine Fehlermeldung angezeigt, sondern es wird mit der nächsten, ausführbaren Anweisung weiter gearbeitet. Das ist insofern von Bedeutung, als der Programmierer die notwendigen Plausibilitätsprüfungen selbst durchführen muß, wenn er nicht Opfer von Fehleingaben oder Rechenfehlern werden will.

Fehlernummer: 6

Fehlermeldung: `Overflow in` ⟨Zeilennummer⟩

Bedeutung: Überlauf.

Erläuterung:

Liegt das Ergebnis einer Berechnung durch BASIC außerhalb des zulässigen Bereichs, so wird diese Fehlermeldung als Warnung ausgegeben, ohne jedoch einen Programmabbruch zu verursachen.

Beispiel:

```
10 A=1E+38
20 PRINT A+A
30 PRINT "Programmende"
```

Ergebnis:

```
Overflow
1.70141E+38
Programmende
Ready
```

Fehlernummer: 7

Fehlermeldung: `Memory full in` ⟨Zeilennummer⟩

Bedeutung: Der Speicher des ⟩⟩ CPC 464 ⟨⟨ ist voll.

Erläuterung:

Diese Fehlermeldung bedeutet, daß der Speicher des ⟩⟩ CPC 464 ⟨⟨ vollständig gefüllt ist. Dies kann folgende Ursachen haben:

1) Das Programm ist zu groß.

2) Das Programm enthält zu viele **FOR-NEXT**-Schleifen.

3) Das Programm enthält zu viele Verschachtelungen von 3 Unterprogrammen.

4) Im Programm treten zu viele Variablen auf.

5) Durch die MEMORY-Anweisung wurde der für BASIC verfügbare Speicherplatz zu klein bemessen.

Hinweis:

Ein geöffneter Kassettenfile benötigt einen Puffer im Speicher des ⟩⟩ CPC 464 ⟨⟨, was dazu führen kann, daß die Werte für **MEMORY** eingeschränkt sind.

Fehlernummer: 8

Fehlermeldung: `Line does not exist in` ⟨Zeilennummer⟩

Bedeutung: Eine angesprochene Zeilennummer existiert nicht.

Erläuterung:

Die angesprochene Zeilennummer ist im Programm nicht vorhanden. Fehlermeldungen dieser Art kommen vor, wenn zu einer nicht vorhandenen Zeile verzweigt wird. Sie erscheint häufig im Zusammenhang mit `GOTO` oder `GOSUB`, oder auch bei `IF THEN ELSE`.

Fehlernummer: 9

Fehlermeldung: `Subscript out of range in` ⟨Zeilennummer⟩

Bedeutung: Ein Index liegt außerhalb des festgelegten Bereichs.

Erläuterung:

Diese Fehlermeldung bedeutet, daß ein Index außerhalb des zulässigen Bereichs liegt. Sie wird ausgegeben, wenn bei der Dimensionierung eines Feldes mit der DIM-Anweisung zu kleine Werte gewählt wurden.

```
10 DIM A(2)
20 A(4)=55
```

Ergebnis: `Subscript out of range in 20`

Vergleichen Sie hierzu: `DIM`

Fehlernummer: 10

Fehlermeldung: `Array already dimensioned in` ⟨Zeilennummer⟩

Bedeutung: Eine Feldvariable wurde bereits dimensioniert.

Erläuterung:

Diese Fehlermeldung besagt, daß bereits ein Feld mit gleichem Namen dimensioniert wurde. In diesem Fall muß entweder ein anderer Variablenname gewählt werden oder das ursprüngliche Feld gelöscht werden.

Eine wiederholte Dimensionierung kann auch "verdeckt" vorkommen, so daß es dem Programmierer anfänglich gar nicht auffällt. Hierzu folgendes Beispiel:

```
100 FOR I=0 TO 10
110 A(I)=I*I
120 PRINT A(I);
130 NEXT I
140 DIM A(100)
150 PRINT "Neuer Abschnitt"
```

Ergebnis:

```
0 1 4 9 16 25 36 49 64 81 100
Array already dimensioned in 140
```

Im ersten Programmteil (Anweisungsnummern 100 bis 130) wird ein Feld mit 11 Elementen bearbeitet, und zwar von A(0) bis A(10). Damit ist die intern vorgegebene Dimensionierung ausgenutzt. In der Anweisungsnummer 140 wird die Variable erneut, und damit unzulässigerweise ein zweites Mal dimensioniert. Das Ergebnis ist die oben angezeigte Fehlermeldung:

`"Array already dimensioned in 140".`

Vergleichen Sie hierzu: `DIM, ERASE`

Fehlernummer: 11

Fehlermeldung: `Division by zero in` ⟨Zeilennummer⟩

Bedeutung: Es wurde durch Null dividiert.

Erläuterung:

Eine Division durch Null "0" ist nach den Regeln der Mathematik nicht zulässig und wird deshalb auch durch BASIC als mathematischer Fehler angezeigt.

Beispiel:

```
10 INPUT WERT
20 PRINT WERT/ZAHL
30 PRINT "Programm laeuft weiter"
40 END
```

Ergebnis:

```
? 78
Division by zero
1.70141E+38
Programm laeuft weiter
Ready
```

Fehlernummer: 12

Fehlermeldung: `Invalid direct command in` ⟨Zeilennummer⟩

Bedeutung: Diese direkte Eingabe ist im Kommandomodus unzulässig.

Erläuterung:

Es wird im Direkt-Modus eine BASIC-Anweisung eingegeben, die nur in einem Programm angewendet werden darf.

Beispiel:

```
DEF FN C(A,B)=A*A + B*B
Invalid direct command
```

Fehlernummer: 13

Fehlermeldung: `Type mismatch in` ⟨Zeilennummer⟩

Bedeutung: Die Zuweisung zu einer Variablen ist nicht typgerecht.

Erläuterung:

Bei einer Zuweisung von Variablen stimmen die Typen nicht überein und führen zur entsprechenden Fehlermeldung.

Beispiel 1:
```
A$=3
Type mismatch
Ready
```

Beispiel 2:
```
LINE INPUT A
Type mismatch
Ready
```

Bei einer LINE INPUT-Anweisung muß eine Zeichenkettenvariable folgen.

Vergleichen Sie hierzu: `DEFINT, DEFSTR, DEFREAL`

Fehlernummer: 14

Fehlermeldung: `String space full in` ⟨Zeilennummer⟩

Bedeutung: Zeichenkettenbereich ist belegt.

Erläuterung:

Es wurden zuviele Zeichenketten definiert, so daß der für Zeichenketten reservierte Speicherbereich auch nach einer "garbage collection" nicht groß genug ist. "Garbage collection" bedeutet ein Aufräumen des Arbeitsspeichers.

Die für Zeichenketten reservierte Speicherbereiche werden gelöscht, wenn diese nicht mehr benötigt werden. Das Aufräumen wird vom BASIC-Interpreter selbst durchgeführt.

Beachten Sie, daß während einer "garbage collection" weder Eingaben noch Berechnungen durchgeführt werden können.

Vergleichen Sie hierzu: `FRE`

Fehlernummer: 15

Fehlermeldung: `String too long in` ⟨Zeilennummer⟩

Bedeutung: Eine Zeichenkette besitzt mehr als 255 Zeichen.

Erläuterung:

In einem Programmablauf kann es durch Aneinanderhängen (Addition) von Zeichenkettenausdrücken zu dieser Fehlermeldung kommen.

Fehlernummer: 16

Fehlermeldung: `String expression too complex in` ⟨Zeilennummer⟩

Bedeutung: Ein Zeichenkettenausdruck ist zu kompliziert.

Erläuterung:

Diese Fehlermeldung wird ausgegeben, wenn ein Stringausdruck für BASIC zu kompliziert ist. Dieses tritt ein, wenn zu viele Verschachtelungen enthalten sind.

In einem derartigen Fall sollte der Ausdruck aufgeteilt werden, so daß sich kleinere, für BASIC überschaubare Ausdrücke ergeben.

Fehlernummer: 17

Fehlermeldung: `Cannot CONTinue in` ⟨Zeilennummer⟩

Bedeutung: Der Programmablauf kann nicht fortgesetzt werden.

Erläuterung:

Diese Fehlermeldung tritt nur beim Befehl CONT auf, der z.B. nach `STOP` oder nach einem Fehler eine Fortsetzung des Programms bewirkt. Eine Programmfortsetzung ist jedoch nicht immer möglich.

Dies ist immer dann der Fall, wenn das Programm nach einem Abbruch geändert wurde.

Vergleichen Sie hierzu: `CONT, STOP`

Fehlernummer: 18

Fehlermeldung: `Unknown user function in` ⟨Zeilennummer⟩

Bedeutung: Eine aufgerufene Funktion wurde noch nicht definiert.

Erläuterung:

Wird eine vom Benutzer zu definierende Funktion mit FN aufgerufen, ohne daß diese zuvor mit `DEF FN` definiert wurde, so kann die entsprechende Berechnung nicht durchgeführt werden und es wird die entsprechende Fehlermeldung ausgegeben.

Vergleichen Sie hierzu: `DEF FN`

Fehlernummer: 19

Fehlermeldung: `RESUME missing in` ⟨Zeilennummer⟩

Bedeutung: In einer Fehlerbehandlungsroutine fehlt die RESUME-Anweisung.

Erläuterung:

Im Zusammenhang mit der `ON ERROR GOTO`-Anweisung muß eine Fehlerbehandlungsroutine mit RESUME abgeschlossen werden. Fehlt die RESUME-Anweisung, so ist eine Wiederaufnahme des regulären Programmablaufs nicht möglich.

Vergleichen Sie hierzu: `RESUME, ON ERROR GOTO, ERR, ERL`

Fehlernummer: 20

Fehlermeldung: Unexpected RESUME in <Zeilennummer>

Bedeutung: RESUME steht nicht innerhalb einer Fehlerbehandlungsroutine.

Erläuterung:

Eine RESUME-Anweisung ist nur innerhalb einer Fehlerbehandlungsroutine zulässig, die mit ON ERROR GOTO vordefiniert sein muß. Ein Einsprung in eine Fehlerbehandlungsroutine ist meist ein Programmierfehler wie in dem folgenden Programmbeispiel:

```
10 ON ERROR GOTO 90
20 DIM A(20)
30 FOR I=1 TO 15
40 A(I)=I
50 PRINT A(I)
60 GOTO 90:REM *** Programmfehler ***
70 NEXT
80 END
90 ON ERROR GOTO 0
100 PRINT ERR,ERL:RESUME 100
```

Ergebnis:

```
1
20            100
Unexpected RESUME in 100
```

Vergleichen Sie hierzu: RESUME, ON ERROR GOTO, ERR, ERL

Fehlernummer: 21

Fehlermeldung: `Direct command found`

Bedeutung: Direkt-Kommando beim Laden von Kassette gefunden.

Erläuterung:

Sollen mit `LOAD` Programme vom Datacorder geladen werden und wird eine Zeile ohne Zeilennummer erreicht, so wird diese als Direktanweisung interpretiert und die Fehlermeldung `Direct command found` ausgegeben. In der Regel sind in einem derartigen Fall die gelesenen Daten kein Programm, sondern Texte.

Fehlernummer: 22

Fehlermeldung: `Operand missing in` ⟨Zeilennummer⟩

Bedeutung: Es fehlt in einer Anweisung ein Operand.

Erläuterung:

Diese Fehlermeldung tritt auf, wenn in einem Ausdruck, bei einem Funktionsaufruf oder auch bei Anweisung ein Operand fehlt. Die Fehlermeldung tritt nicht nur in einem Programm, sondern auch im Direktmodus auf.

Beispiel:

```
PRINT 3*4.75+
Operand missing
Ready
```

Fehlernummer: 23

Fehlermeldung: `Line too long in` ⟨Zeilennummer⟩

Bedeutung: Eine Zeile ist zu lang.

Erläuterung:

Vom Benutzer direkt oder im Programm eingegebene Zeilen werden von BASIC in eine eigene Form umgewandelt. Diese kann nur eine bestimmte Größe annehmen, andernfalls wird die Fehlermeldung `Line too long` ausgegeben. Diese Fehlermeldung tritt nur in sehr seltenen Fällen auf. In einem derartigen Fall sollte man eine Zeile in zwei oder mehrere aufteilen, was meist auch der Übersichtlichkeit des Programms dient.

Fehlernummer: 24

Fehlermeldung: `EOF met in` ⟨Zeilennummer⟩

Bedeutung: Das Ende einer Datei wurde bereits erreicht.

Erläuterung:

Wird beim Lesen einer Datei vom DATACORDER nicht das Ende der Datei beachtet, sondern über das Ende der Datei hinaus gelesen, so ist dies nicht möglich und führt zu dieser Fehlermeldung. Das Ende einer Datei kann mit EOF abgefragt und mit einer IF THEN-Anweisung die INPUT-Anweisung im Programmablauf übersprungen werden.

Vergleichen Sie hierzu: `EOF, OPENIN, INPUT, LINE INPUT`

Fehlernummer: 25

Fehlermeldung: `File type error in` ⟨Zeilennummer⟩

Bedeutung: Der Typ einer Datei ist nicht korrekt.

Erläuterung:

Werden Daten, z.B. Text-Dateien, vom DATACORDER eingelesen, so muß die Aufzeichnungsart beachtet werden, d.h., sie können nicht mit der `LOAD`-Anweisung geladen werden. Außerdem können nur Dateien, die im ASCII-Code abgelegt wurden, mit der `OPENIN`-Anweisung gelesen werden.

Vergleichen Sie hierzu: `LOAD, RUN, SAVE, OPENIN, OPENOUT`

Fehlernummer: 26

Fehlermeldung: `NEXT missing in` ⟨Zeilennummer⟩

Bedeutung: Eine `FOR-NEXT`-Schleife ist unvollständig.

Erläuterung:

Wird zur mehrfachen Wiederholung eines Programms eine Schleife mit der dazugehörigen `FOR`-Anweisung begonnen, so muß das Schleifenende mit `NEXT` gekennzeichnet werden.

Die Fehlermeldung tritt insbesondere dann auf, wenn zwar eine `NEXT`-Anweisung vorhanden ist, die angegebene Laufvariable aber nicht mit der in der `FOR-NEXT`-Schleife übereinstimmt. Weiterhin muß bei geschachtelten Schleifen bei Angabe mehrerer Parameter nach `NEXT` auf die richtige Reihenfolge geachtet werden.

Beispiel:

```
10 FOR I=1 TO 10
20 FOR K=1 TO 3
30 PRINT I*K
40 NEXT I
```

Ergebnis:

```
NEXT missing in 10
```

In Zeile 40 wurden versehentlich die Laufvariablen `K` vergessen.

Vergleichen Sie hierzu: `FOR NEXT`

Fehlernummer: 27

Fehlermeldung: `File already open in` ⟨Zeilennummer⟩

Bedeutung: Der angesprochene File ist bereits eröffnet.

Erläuterung:

Beim Ablegen bzw. Lesen von Dateien mit dem DATACORDER ist es nicht möglich, eine `OPENOUT`- oder `OPENIN`-Anweisung mehrfach zu geben, ohne vorher die bereits geöffnete Datei wieder zu schließen. Das Schließen erfolgt mit der `CLOSEIN`- bzw. mit der `CLOSEOUT`-Anweisung. Nach dieser Anweisung kann ein neuer FILE eröffnet werden.

Beispiel:

```
10 OPENOUT "NAMEN"
20 OPENOUT "TELEFONNUMMERN"
```

(weiteres Programm)

führt beim Starten des Programms zur Fehlermeldung: `File already open in 20`

Vergleichen Sie hierzu: `CLOSEIN, CLOSEOUT, OPENIN, OPENOUT`

Fehlernummer: 28

Fehlermeldung: `Unknown command`

Bedeutung: Eine Anweisung ist in BASIC nicht definiert.

Erläuterung:

Diese Fehlermeldung kommt in der Regel bei alleiniger Arbeit mit dem BASIC-Interpreter nicht vor. Sie kann aber bei Kommunikation mit weiteren Geräten auftreten, wenn Anweisungen gegeben werden, die BASIC nicht bekannt sind.

Fehlernummer: 29

Fehlermeldung: WEND missing in ⟨Zeilennummer⟩

Bedeutung: In einer WHILE-WEND-Schleife fehlt das WEND.

Erläuterung:

Beim Programmieren einer WHILE-WEND-Schleife ist ebenso wie bei der FOR-NEXT-Schleife ein Kennzeichnen des Schleifenendes notwendig. Tritt die Fehlermeldung WEND missing auf, so fehlt zu einer WHILE-Anweisung die entsprechende WEND-Anweisung als Schleifenabschluß. Hierbei muß beachtet werden, daß sich ein WEND immer auf das vorhergehende WHILE bezieht und deshalb zu jeder WHILE-Anweisung eine WEND-Anweisung notwendig ist.

Beispiel:

```
10 MODE 1
20 WHILE SUMME<50
30 INPUT "Ihre neue Zahl : ";ZAHL
40 SUMME=SUMME+ZAHL
50 PRINT " SUMME = ";SUMME
60 REM *** WEND VERGESSEN ***
70 END
```

Ergebnis:

Das Programm bricht in Zeile 20 mit der Fehlermeldung WEND missing in 20 ab, da das Schleifenende nicht mit der WEND-Anweisung gekennzeichnet wurde. Dieser Fehler tritt nicht auf, wenn Zeile 60 eingefügt wird:

```
60 WEND
```

Vergleichen Sie hierzu: WHILE WEND

Fehlernummer: 30

Fehlermeldung: `Unexpected WEND in ><Zeilennummer>`

Bedeutung: Bei einer `WEND`-Anweisung fehlt die zugehörige `WHILE`-Anweisung.

Erläuterung:

Diese Fehlermeldung besagt, daß ein durch `WEND` zu kennzeichnendes Schleifenende angetroffen wurde, ohne daß eine dazugehörige Schleife vorhanden ist.

Dem BASIC-Interpreter ist in diesem Fall keine Bedingung vorgegeben, die er beim Erreichen der `WEND`-Anweisung überprüft, bevor BASIC einen Programmteil erneut durchläuft.

Vergleichen Sie hierzu: `WHILE WEND`



DAS STANDARD BASIC-BUCH zum Schneider-Computer CPC 464
E. Unger Best.-Nr. B-201

NEUERSCHEINUNG. Das Standardwerk zum Schneider-Computer **CPC 464!**

Auf ca. 290 Seiten eine klare und verständliche Einführung in die Programmiersprache BASIC mit allen BASIC-Kommandos und einer ausführlichen Beschreibung der Arbeit mit dem Datacorder, dem Cassettenteil des Computers. Das Buch überschüttet den Lernenden nicht mit einer Fülle von Informationen, sondern führt ihn behutsam an den Lernstoff heran. Der Leser wird in die Lage versetzt, unmittelbar am Computer arbeitend, das Gelernte in die Praxis umzusetzen. Die Anleitungen sind durch die Art der Aufgabenstellung und die schrittweise Darstellung der Lösungswege auf die Bedürfnisse des Selbstunterrichts abgestellt.

Das Buch ist praxisbezogen: Schon nach wenigen Kapiteln ist der Leser imstande BASIC-Programme zu verstehen und eigene kleine Programme zu schreiben. Der Einstieg in die neue Materie **„Logik des Programmierens"** wird Ihnen durch sorgfältige Aufbereitung des Lehrstoffes so leicht wie möglich gemacht.

Alle englischen Fehlermeldungen des **CPC464-BASIC** werden in das Deutsche übersetzt und eingehend erläutert. Die möglichen Fehlerursachen werden aufgezeigt und – wenn nötig – durch Programm-Beispiele erklärt.

Ein Spitzenbuch mit über 50 praxisnahen Übungs- und Anwenderprogrammen.

Alle diese Programme finden Sie auf einer **Cassette**, die Ihnen die riesige Mühe des Eintippens erspart. Lesen Sie dazu auch die Beschreibung zur **Programm-Cassette** in diesem Prospekt.

Sie erhalten ein umfassendes und außerordentlich reichhaltiges Fachbuch, das Ihnen das Tor zu Ihrem neuen Computer, dem Schneider **CPC464** öffnet. Ein Buch also, das Sie begeistern wird.

Buch und Cassette im Verbund sind die richtigen Begleiter auf Ihrem Weg zum Experten am Schneider-Computer **CPC464.**

ca. 290 Seiten 68,– DM



DIE PROGRAMM-CASSETTE zum Buch:

DAS STANDARD BASIC-BUCH zum Schneider Computer **CPC464**
E. Unger Best.-Nr. C-204

Die **Programm-Cassette** enthält die Beispielprogramme des **STANDARD BASIC-BUCHS** und ist hervorragend geeignet, Ihnen die praktische Arbeit und das Training am Computer ganz wesentlich zu erleichtern. **PROGRAMM-CASSETTE** laden und mit dem Schneider-Computer üben, das macht den Meister.

Die Palette der Programme geht von einfachen BASIC-Übungsprogrammen bis zu nützlichen Anwenderprogrammen wie zum Beispiel:

* **Sortierprogramm**
* **Programm zur Umrechnung fremder Währungen**
* **Programm zum Erstellen eines beliebigen Zeichenvorrats** (Tasten können beliebig belegt werden)
* **Dienstprogramme** (Hiermit können sämtliche mathematischen, chemischen, physikalischen sowie internationale Zeichensätze mit dem Computer vereinbart und genutzt werden)
* **Textverarbeitung** (Schreiben von Texten, Ablegen und Einlesen eines Briefzeilenprogrammes, Einfügungen, Druckerausgabe)
* **Kartei** (Verwalten von bis zu 200 Adressen, elegante Benutzerführung, Adresseneingabe im Dialog, Sichern der Adressen auf dem Datacorder, Einlesen von gespeicherten Adressdateien, Suchen von Adressen nach frei wählbaren Suchbegriffen, Blättern der Adressdatei, Löschen einzelner Adressen, Neueingabe von Adressen
* Mein Kassenbuch (Einnahmen-Ausgaben Abrechung)
* Mein Telefonbuch (Auswahl aus erfaßten Tel.-Nummern nach Namen und Rufnummern)
* und viele nützliche Programme mehr...

Die Cassette enthält über 50 Programme, die Sie nicht mehr mit riesiger Mühe einzutippen brauchen. Vor allem aber ersparen Sie sich viele Fehler und viele Korrekturen. Sie können **„Gleich zur Sache"** kommen.

Die Cassette hilft Ihnen, schon nach kurzer Zeit BASIC-Programme zu verstehen, selbst zu verändern (erweitern oder kürzen) und so schneller als sonst üblich, eigene kleine Programme zu schreiben.

Auch diese Cassette wird Sie begeistern, sie gehört zu Ihrem Schneider-Computer genauso wie das **STANDARD BASIC-BUCH.**

Weniger als 1,50 DM je Programm

Über 50 Übungs- und Anwenderprogramme 74,– DM

DAS GROSSE BASIC-LEXIKON zum Schneider-Computer CPC464

NACHSCHLAGEWERK UND PROGRAMMSAMMLUNG FÜR ANFÄNGER UND FORTGESCHRITTENE

Erstellt von einem Autorenteam — Best.-Nr. B-203

NEUERSCHEINUNG. Mit der Erfahrung viele Programmiererjahre hat das Autorenteam für Sie ein Nachschlagewerk von besonderer Qualität geschaffen.

Im **GROSSEN BASIC-LEXIKON** finden Sie den gesamten umfangreichen Befehlssatz (ca. 180 Befehle und Funktionen) des Schneider-Computers **CPC464**. Es kommt aber noch viel mehr hinzu.

Alle Befehle und Funktionen sind nach einem klaren, leicht verständlichen Schema in alphabetischer Ordnung wie folgt dargestellt:

1. **BASIC**-Schlüsselwort
2. **FORMAT** (Schreibvorschrift))
3. **ZWECK** (Wofür wird das Schlüsselwort benötigt)
4. **ANWENDUNG** (Ausführliche Erläuterung des Schlüsselwortes und Vorbereitung des Programmbeispiels)
5. **PROGRAMM-BEISPIEL**
6. **ERGEBNIS** (Das Ergebnis des Beispielprogramms wird – wenn nötig – besprochen)
7. **VERGLEICHSHINWEISE** (Es wird auf vergleichbare Schlüsselwörter verwiesen)

Dieses Nachschlagewerk hat also neben seinem Charakter als Lexikon noch den unschätzbaren Vorteil, auch eine wertvolle Programmsammlung zu sein mit einem hohen Nutzwert für Sie als Programmierer.

Sie können zu jedem Zeitpunkt eingehende Informationen zum gewünschten Schlüsselwort nachschlagen.

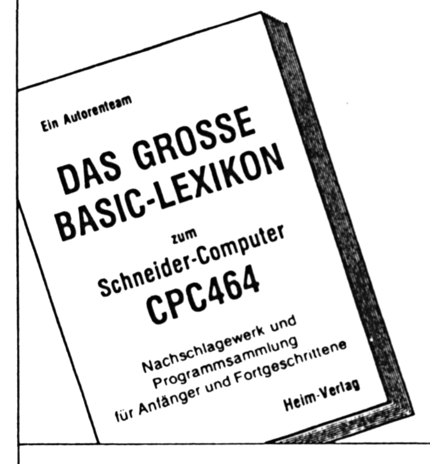
Eine Stärke des Buches sind die auf das Schlüsselwort ausgerichteten Programm-Beispiele, denn dort finden Sie – eingebettet in das Programm – die Art und Weise, wie das Schlüsselwort zu benutzen ist.

Deshalb ist beim Lernen und Erstellen eigener Programme für Ihren Schneider **CPC 464 DAS GROSSE BASIC-LEXIKON** eine wertvolle und unerläßliche Hilfe.

Alle diese Programme finden Sie ebenfalls auf einer **PROGRAMM-CASSETTE**. Damit haben Sie auch hier wieder die Möglichkeit auf bequeme Art und Weise am Schneider Computer zu üben und zu lernen.

Eine wertvolle Buchseite für weniger als 30 Pfennige

ca. 200 Seiten — 58.– DM



DIE PROGRAMM-CASSETTE zum Buch:

DAS GROSSE BASIC-LEXIKON zum Schneider-Computer CPC 464

Erstellt von einem Autorenteam — Best.-Nr. C-205

DIE PROGRAMM-CASSETTE enthält die Beispielprogramme des **GROSSEN BASIC-LEXIKONS** und ist hervorragend geeignet, Ihnen die praktische Arbeit und das Training am Schneider-Computer ganz wesentlich zu erleichtern.

Über 150 Programmbeispiele erläutern Ihnen am Bildschirm, wie jedes Programm-Schlüsselwort sachgerecht in entsprechende Programm-Anweisungen eingebracht wird.

Sie haben somit die Möglichkeit in alphabetischer Ordnung gezielt auf das gewünschte Schlüsselwort zuzugreifen und dort das geeignete Beispiel für Ihren Anwendungsfall zu finden.

In dem der Cassette beigefügten Bedienungsheft finden Sie Vorschläge für die Nutzung der Cassette beim Lernen und Üben. So enthält dieses Heft ausführliche Hinweise zur gezielten Wiederholung des Lernstoffes in zusammengehörigen Gruppen verwandter Schlüsselwörter.

Auch diese Cassette ist eine wertvolle Ergänzung Ihrer Lernmittel aus dem **Heim-VERLAG** zum Schneider-Computer **CPC464**.

Weniger als 50 Pfennige je Programmbeispiel.

Über 150 Programmbeispiele — 74.– DM



R. Grum / R. Hund

DAS GROSSE BASIC-LEXIKON zum



Eine Edition aus dem Heim-Verlag

R. Grum
R. Hund

DAS GROSSE BASIC-LEXIKON · Schneider-Computer CPC464

Heim


AMSTRAD
CPC
OCR
MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA
ACME
https://acpc.me/